

AD-A138 009

A SYNTAX DIRECTED EDITOR ENVIRONMENT(U) AIR FORCE INST
OF TECH WRIGHT-PATTERSON AFB OH SCHOOL OF ENGINEERING
J R KOSLOW 05 DEC 83 AFIT/GCS/MA/83D-3

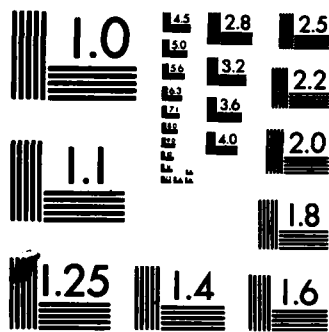
1/2

J R KOSLOW 05 DEC 83 AFIT/GCS/MA/83D-3

UNCLASSIFIED

F/G 9/2

NL



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

AD A138009

DTIC FILE COPY



A SYNTAX
DIRECTED EDITOR
ENVIRONMENT

THESIS

AFIT/GCS/MA/83D-3 John R. Koslow
2Lt USAF

DTIC
ELECTE
S FEB 21 1984

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

DISTRIBUTION STATEMENT A

Approved for public release;
Distribution Unlimited

84 02 17 063

AFIT/GCS/MS/83D-3

Accession For	
NTIS GRA&I	<input checked="checked" type="checkbox"/>
DTIC TAB	<input type="checkbox"/>
Unannounced	<input type="checkbox"/>
Justification	
By	
Distribution/	
Availability Codes	
Dist	Avail and/or Special
A/1	

A SYNTAX
DIRECTED EDITOR
ENVIRONMENT

THESIS

AFIT/GCS/MA/83D-3 John R. Koslow

2Lt USAF

DTIC
ELECTE
S FEB 21 1984 D

Approved for public release; distribution unlimited.

D

A SYNTAX
DIRECTED EDITOR
ENVIRONMENT

THESIS

PRESENTED TO THE FACULTY OF THE SCHOOL OF ENGINEERING
OF THE AIR FORCE INSTITUTE OF TECHNOLOGY
IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS FOR THE DEGREE OF
MASTERS OF SCIENCE IN COMPUTER SCIENCE

by

John R. Koslow
2Lt USAF
Graduate Computer Science

5 December 1983

Approved for public release; distribution unlimited.

PREFACE

This document describes the implementation and modification of a software development environment for a medium size computer system based on a syntax directed editor. It was developed for use in writing programs in the ADA programming language, but can be used for any language which can be described using extended BNF notation.

This effort is a follow on effort to work done by Captain Scott E. Ferguson at the Air Force Institute of Technology. It consists of modifications and extensions to the prototype environment he developed. His work is cited in the bibliography.

I would like to thank my advisors for their high level of expectation which required my best effort to satisfy. Thanks also to my special friends who put up with a person who had nothing but computers on his mind, and to my parents who have always given me support.

John R. Koslow

ABSTRACT

↓

This document describes the implementation and modification of a software development environment for a medium size computer based on a syntax directed editor. Although it was developed for use with the ADA programming language, most of the environment is driven by a language syntax description, and can therefore process virtually any current or future programming language. This environment is an extension of a prototype developed previously at the Air Force Institute of Technology.

↑

TABLE OF CONTENTS

1. Introduction	1
1.1. Syntax Directed Editor	1
1.2. Other Tools in the Environment	2
1.3. A Look Forward	2
2. Syntax Directed Editors	4
2.1. Why syntax directed editors?	4
2.2. SYNDE	5
3. System Design	7
3.1. Overall Environment	7
3.2. Current Development	7
3.3. Long Range Goals	8
3.4. User Friendliness	8
4. System Development	9
4.1. Assembling the source code	9
4.2. Integration of display package	10
4.3. Interfaces between tools	11
4.4. Tool Independence	11
4.5. Individual tools	12
5. Syntax Tree Background	13
5.1. BNF Syntax Description	13
5.2. The Syntax Tree	14
5.3. Program tree display	14
5.3.1. The Extended Cursor	15
5.3.2. Display Selection	18
5.3.3. Display implementation	21
6. SYNDE	24
6.1. Program tree synthesis	24
6.1.1. Creating the tree	24
6.1.2. Modifying the tree	32
6.2. Tool development	34
7. META	36
7.1. Tool development	36
7.2. Meta Syntax Description	36
7.3. Language Subsets	37
7.4. META Conversions	38
8. Other Tools	40
8.1. Config	40
8.1.1. Tool Development	40
8.1.2. General information	41
8.1.3. Input command sequences	41

8.1.4. Output command sequences	42
8.2. Lister	42
9. Conclusions and Recommendations	45
9.1. Conclusions	45
9.2. Recommendations	46
9.2.1. Conversion of code to ADA	46
9.2.2. Incremental compilation	46
9.2.3. Multi-tasking	47
9.2.4. Program documentation	47
9.2.5. User interface improvements	48
BIBLIOGRAPHY	50
Appendix I. META SYNTAX DESCRIPTION LANGUAGE	51
Appendix II. META DESCRIPTION FOR ADA	53
Appendix III. META DESCRIPTION FOR ADA0	74
Appendix IV. SYSTEM USER MANUAL	78
IV.1. Introductory Manual	78
IV.2. Syntax Directed Editor Manual	82
IV.3. Terminal Configuration Manual	89
IV.4. Meta Preprocessor	92
Appendix V. GLOSSARY	99

LIST OF FIGURES

Figure 5-1:	Point cursor display	16
Figure 5-2:	Extended cursor display	17
Figure 5-3:	Focus at "decl" node	18
Figure 5-4:	Display after "right" command,	18
Figure 5-5:	Two nested procedures	20
Figure 5-6:	Elided inner procedure	20
Figure 5-7:	Display of outer procedure	21
Figure 5-8:	Display of proc_body	22
Figure 6-1:	ADA syntax tree root	25
Figure 6-2:	Initial expansion of root	26
Figure 6-3:	Tree after selection of proc_body	27
Figure 6-4:	Tree after expansion	29
Figure 6-5:	Display of tree	29
Figure 6-6:	Proc_body subtree	30
Figure 6-7:	Identifier subtree and display	31
Figure 6-8:	Identifier subtree after selection	32
Figure 6-9:	Proc_body subtree	33

INTRODUCTION

1. INTRODUCTION

The Air Force spends a large amount of time and money on the development of computer software. The development of a user friendly integrated software development environment can greatly reduce both the time and cost of software development. This research effort attempts to integrate a number of software tools which are presently available into a user friendly integrated system for the UNIX operating system. This should help to reduce the software development costs for the Air Force and the Department of Defense.

Computers have been changing the way people work for many years. One area which has been extremely slow to benefit from computer assistance is the development of computer software. Only recently have environments been developing which utilize the power of the computer in the software development stage.

1.1. SYNTAX DIRECTED EDITOR

Computers, as well as other machines, have been used to eliminate the repetitive, mistake prone steps in many production areas. Such a mistake prone step exists in the production of computer software. The generation of syntactically correct code seems to be a major problem in even the best of programmers. This has led to the recent work on syntax directed, or language oriented, editors. A syntax directed editor permits the construction of only those programs which are syntactically correct. Program elements are selected by the user, then required words and punctuation are automatically handled by the editor. Only those areas which are variable need to be entered by the programmer. This results in syntactically correct programs and leads to many other benefits. The compiler can be simpler and faster because it no longer need check the syntax of the program being compiled. And the programmer

INTRODUCTION

can concentrate on logical and semantic problems without the worry of inadvertent syntax errors. Syntax directed editors, and the environments built around them, are still in the early stages of development. Work is being done at many computer facilities to develop a good syntax directed editor. A prototype syntax directed editor, and an environment built around it, was developed by Captain Scott Ferguson at the Air Force Institute of Technology in 1982. A large part of this research effort is based on the groundwork laid out by Captain Ferguson's work on a syntax directed editor, which is discussed in the next chapter.

1.2. OTHER TOOLS IN THE ENVIRONMENT

The editor is only the central tool in a well developed programming environment. Screen manipulation packages are tied in to the editor to take advantage of the capabilities and varying abilities of different types of terminals. Control can pass from the editor to a compiler or interpreter, then return to the editor. Some means of listing the program in a form familiar to the programmer must also be available. All of these functions must exist in the environment in such a way that the movement between the tools, as well as commands within each one, is easy to understand by anyone who would have reason to use the system.

1.3. A LOOK FORWARD

This research effort attempts to move forward in the development of such an integrated software development system. The main tool of the environment is a syntax directed editor. Tools which have been previously developed on various systems have been brought together and tied into the syntax directed editor to form the basis of the integrated system. Individual tools and interfaces have been modified to increase the ease of use for both experienced and less experienced programmers.

INTRODUCTION

This paper discusses the overall design of the integrated software development system and the development work performed in this effort. Chapter 2 discusses previous work on syntax directed editors, and some basic information needed to understand them. Chapter 3 discusses the design of the current environment, while chapter 4 discusses the development steps in the current effort. Chapter 5 looks at the "syntax tree", the basic structure on which the editor operates, as well as how such a tree is displayed to the user. Chapter 6 deals with the syntax directed editor, while chapters 7 and 8 discuss the other tools in the environment. The final chapter provides conclusions and recommendations for future research efforts in this area. The appendices provide important reference material, particularly the user manuals contained in appendix IV.

2. SYNTAX DIRECTED EDITORS

Work on the development of syntax directed editors, and the environments in which they will be used, is still in the early stages. This chapter will look at why development work is being done in this area, and describe the previous work on which this thesis effort is based.

2.1. WHY SYNTAX DIRECTED EDITORS?

As mentioned in the first chapter, the area of software development has been an area which has not profited from the use of computers. Programmers have been writing and entering programs in much the same way since the development of higher order languages. All the work was done by the programmer, both in writing and entering the program. Very little work was done to utilize the power of the computer in developing software. Software was developed away from the computer, then submitted to the computer which would indicate where the errors were.

As systems became more interactive, and video displays more common, it was natural to try to find ways in which the power of the computer could be used to assist in the development of computer software. One area which could benefit from this is the generation of syntactically correct code.

Writing syntactically correct code has been a problem for even the best of programmers. Proper placement of separators or delimiters such as semicolons or "begin" and "end" is one example of the syntactic problems encountered by most programmers. The computer must check the program for syntactic correctness at some point, so rather than check at compile time it could check during the editing session. This produces the idea of syntax directed, or language oriented editors.

Syntax directed editors guarantee that the program entered is syntactically

SYNTAX DIRECTED EDITORS

correct based on the syntax description of the language in which it is being written. While not eliminating semantic or logical errors, the editor does free the user from worrying about syntax errors. Thus the user can concentrate on understanding the program and solving the other types of errors.

Work on syntax directed editors is taking place at many institutions including Carnegie -Mellon University (Ref 7,10) and the Air Force Institute of Technology (AFIT)(Ref 8). This thesis effort is based on work done at AFIT by Captain Scott E. Ferguson (Ref 8).

2.2. SYNDE

Captain Ferguson's work resulted in a prototype editor environment based on a syntax directed editor called SYNDE (SYNtax Directed Editor). Much of his work was based on research by Bruce J. MacLennan of the Naval Postgraduate School (Ref 9) and two of Mister MacLennan's former thesis students, William R. Shockley and Daniel P. Haddow (Ref 11). SYNDE was developed for use on a microcomputer, and was written in the programming language C.

The central tool in the SYNDE environment is the syntax directed editor itself. The editor builds a tree based on the language syntax description, and manipulates this structure. All editing commands affect the tree structure and it's elements, unlike conventional editors which affect the individual characters. This will be discussed in more detail in a later chapter.

Other tools in the environment include a program lister, terminal configuration program, and a syntax description processor called META. All of these tools, including the editor, were written for use on microcomputers. This

SYNTAX DIRECTED EDITORS

thesis effort involved transferring these tools to a larger machine, modifying them to run on the larger machine, and then modifying the individual tools to expand the capabilities of the overall editor environment.

SYSTEM DESIGN

3. SYSTEM DESIGN

The goal of this research effort is to further the development of an integrated software development environment. It is being designed and tested with the ADA programming language as the target language for the environment to handle. However it is being developed as a general purpose package which will require only minor adjustments to work with other programming languages. The creation of a new syntax description would be the only modification needed for most of the tools due to the language independence of those tools. The language dependent tools would have to be developed for the particular language desired, but could be easily integrated.

3.1. OVERALL ENVIRONMENT

The central thrust is to integrate the various software development tools into a single package so that the entire software development effort can take place from within the editor. It will provide a working environment for software development tailored to that particular application and which is one step removed from the operating system on which it is run. Once the editor is entered code can be generated, compiled, listed, debugged, or reedited without leaving the editor environment, because all of the tools necessary to perform these actions are accessible from the editor. Other programs or system abilities which are not needed for software development cannot be accessed from the editor environment, restricting the user to those tools useful for software development.

3.2. CURRENT DEVELOPMENT

The current effort involves the development of a working version of this type of system on a medium size computer such as the VAX 11/780. The tools to be used involve a syntax directed editor, source program lister, compiler, and an interpreter. Other programs including a system configuration preprocessor

SYSTEM DESIGN

and language description preprocessor will be part of the environment. These programs will in turn depend on lower level packages such as a screen display package which will allow the use of various types of screen display devices independent of the editor. The compiler and interpreter are being developed as a separate research effort and will be integrated into the environment when they become available.

3.3. LONG RANGE GOALS

Long range design goals include the ability to take advantage of multi-tasking or background mode running of tools such as the compiler to reduce the amount of time the user spends sitting waiting for the work to be done. Steps are being taken to provide the necessary information for such an expansion, but this is not an issue of the current research.

3.4. USER FRIENDLINESS

An issue of importance throughout this effort is the design of an environment which is friendly to the user. An environment which is not friendly will frequently not be used and will thus be of little importance regardless of how efficient it may be. Issues of user friendliness are considered at all levels of this effort. From the types of commands available, to how those commands respond, to the movement between software tools, the ease of use must be considered. Some decisions will be made that will result in a system that seems less than perfect to the user, but these can only be made after careful consideration of the tradeoffs involved. In the long run the environment is being developed for the user, and thus should be as friendly as possible.

SYSTEM DEVELOPMENT

4. SYSTEM DEVELOPMENT

The development of the current syntax directed editor environment involved a number of steps. The first step in developing the system was to assemble working versions of all of the desired tools on the target host computer. The decision was made to develop the environment for a VAX 11/780 running under the UNIX operating system, due to the system's availability at AFIT (Air Force Institute of Technology), as well as the wide spread use of this or comparable systems throughout the Department of Defense. Next was the integration of a display handling package into the environment to provide a more terminal independent environment. Work was then done on the interface between the various tools, and determining the areas of dependence between the tools. Finally, modifications were made to individual tools to improve their performance, and thus the performance of the entire environment.

4.1. ASSEMBLING THE SOURCE CODE

When the source code for all of the tools was assembled on the VAX a number of problems became evident. The major problem was that SYNDE, the syntax directed editor, was not written in a standard form of the programming language C. It was written in a dialect available on many microcomputers, but this dialect was not portable to the VAX. Many of the problems were solved by relatively minor changes, but these were difficult to detect. Other problems required the generation of the code for functions which were predefined in the dialect used, but were not standard C functions. These two types of problems were very time consuming even though the actual code generated was not excessive.

Other changes were needed to make the program compatible with the system calls of the UNIX operating system instead of the microcomputer system on which the editor was originally developed. Thus the task of assembling

SYSTEM DEVELOPMENT

working versions of all of the required tools turned out to be a major project.

4.2. INTEGRATION OF DISPLAY PACKAGE

Once the code was working, the modifications could begin. The first step was the integration of the display handling package into the syntax directed editor. The editor had a number of display capabilities but these were all dependent on the user entering the correct set of keystrokes for his particular terminal when he configured the system. The user was required to know and enter the often complex sequences to move the cursor, set highlight mode, and other similar capabilities for the particular terminal being used. A change in the terminal being used required the user to learn the sequences for the new terminal and enter them using the Config program. On a large system the number of different types of terminals available to the user can be enormous, especially if dial-up capabilities are available. The system often has access to a database describing these various terminals and their capabilities. The UNIX operating system, under which this environment was developed, has access to such a database. Thus it is unnecessary, and inadvisable, to require the user to enter the control sequences for the various output commands. This is another example of where the power of the machine should be used in order to simplify the use of the environment as well as to reduce the chance of errors in the information needed. The UNIX "CURSES" (Ref 1) display package has access to the required database and was integrated into the syntax directed editor to provide a more terminal independent environment. The ability to enter the codes for a particular terminal and store them in a local file, even if the terminal type is not in the database, is a useful feature which deserved to be retained.

SYSTEM DEVELOPMENT

4.3. INTERFACES BETWEEN TOOLS

The editor is the main tool of the environment being developed, and thus a means of going from the editor to other tools, and returning, is important. Therefore work on the interface between the tools was needed. The ability to generate system calls from within the editor program was used to invoke the other environment tools such as the lister and the compiler. As new tools are added, modifications must be made both in the editor and in the new tools. The appropriate system call must be generated by the editor when the tool is requested by the user. This is done by setting the parameters for the general system call to those required for a call to the particular tool. New tools having more than one argument require modification to the general system call to allow the longer argument list. Modifications must be made in the new tool to ensure that a call is made to return control to the editor if the tool was invoked from the editor, or to properly exit if called from some other place. The return to the editor will be the usual case and is accomplished by a system call to the editor with the appropriate arguments. If the tool was not called from the editor it should return to the operating system. This allows the tools to be general purpose, and they can be used outside of the editor environment if necessary. Although use of the tools within the editor environment is the preferred use, making them general purpose provides them with much greater flexibility.

4.4. TOOL INDEPENDENCE

Once all of the tools were working and tied together to form a basic software development environment, changes would be made to the individual tools. Therefore time was spent examining the effect that changing one tool would have on the other tools and on the overall environment. Improvements and additions to individual tools should be allowed, but these should not affect

SYSTEM DEVELOPMENT

the operation of the overall software development system in any significant way. One way to accomplish this goal is to have the various tools perform only a specific function, and not depend on an implementation used in any of the other tools. Only a few minor changes were needed to improve this area. The tools were then fairly independent and as the modifications were made to the tools to improve their individual performance, the integrity of the development system was maintained.

4.5. INDIVIDUAL TOOLS

Once a working version of the editor environment was available on the UNIX system, work could begin on the desired modifications to the individual tools. Some of these were modifications to the interfaces between tools, while others were changes to the basic tool itself such as adding new commands or tying in new packages.

The next chapter discusses some of the background material on the "syntax tree", which is the basic structure on which the individual tools operate. This is followed by chapters covering the capabilities and uses of the individual tools which have been integrated into the software development package up to this point.

BACKGROUND

5. SYNTAX TREE BACKGROUND

The syntax tree is the basic structure on which all of the tools in the environment operate. This tree is derived from the textual description of the language provided to the tools. This chapter discusses a general form of textual syntax descriptions, what the tree is, and how it is displayed to the user.

5.1. BNF SYNTAX DESCRIPTION

One common form of describing the syntax of a language is extended BNF (Backus -Naur Form). Extended BNF consists of a sequence of production rules which define the goal symbol in terms of other terminal and non-terminal symbols. Each non-terminal symbol is defined somewhere in the sequence of productions until all symbols have been defined. Production rules are of the following form:

```
non-terminal ::=
    production_for_non-terminal;
```

The production can be either a concatenation or an alternation. A concatenation is a series of terminals or non-terminals which appear in the order specified in the production. An alternation is a choice where only one of the indicated alternatives may be present. The individual alternatives are separated by a vertical bar ("|").

Two other types of elements must be discussed. These are the option and the repeater. Options are elements which may appear in the production either zero or one time. They are indicated by surrounding the element or elements in square brackets ("[" and "]"). Repeaters are elements which may appear zero or more times. These are indicated by surrounding the elements in braces ("{" and "}"). Collectively options and repeaters are referred to as "conditionals", those elements whose presence are not required for syntactic correctness.

BACKGROUND

5.2. THE SYNTAX TREE

The basic structure which the editor deals with is the syntax tree. A syntax tree is a tree structure which represents the program being edited by the user. The root of the tree is the language goal symbol, while the leaves are the elements which correspond to the program as it stands at any particular time. Intermediate nodes represent non-terminal elements of the language definition which have already been expanded. Each node of the tree corresponds to exactly one terminal or non-terminal symbol in the language syntax definition.

As productions are applied to the nodes of the tree, children are created. As discussed in section 5.1, productions can be of two forms : concatenations and alternations. A concatenation production is a series of terminal or non-terminal elements which are to appear in the order specified in the production. In a node corresponding to a concatenation production, a child of the parent node is created for each element of the concatenation. An alternation production consists of a set of available choices, only one of which is chosen. For a node corresponding to an alternation node, only one child is created, and it is created when selection of a single element is made. As will be discussed in section 6.1.1, the terminal character strings corresponding to the reserved words and delimiters in the language are not stored as part of the syntax tree. Only those elements which can be changed are part of the syntax tree.

5.3. PROGRAM TREE DISPLAY

The enormous difference in the structure handled by a syntax directed editor compared to a text editor presents a number of problems in displaying the information in a form which is useful to the user. Thoughtful solutions to these problems can result in a number of advantages which will offset the problems. This section contains a discussion of the problems and their solutions, followed by discussion of the integration of a screen handling package to

BACKGROUND

improve the user interface. The reader should not be concerned with how the example syntax trees are originally obtained. This will be discussed in detail in the next chapter. That information is not necessary for an understanding of the display information.

5.3.1. THE EXTENDED CURSOR

Movement within a conventional text-based screen-oriented editor is usually accomplished by the use of commands such as up, down, left, and right. Such movements place the cursor at a particular cell in which characters may be added, deleted, or changed. This works well because each cell represents both a unit of change and a unit of movement. Both are based on a one character per unit measurement basis. Movement one unit to the right corresponds to moving to the next character of the text, as well as moving to the next cell of the display. The new cell contains that single character, thus the one-to-one correspondence is maintained.

The syntax directed editor, however, wants to deal in program components which are usually longer than a single character. Thus conventional screen movements are inappropriate for the syntax directed editor and result in an extremely complex mapping function from screen coordinates to the structure elements if attempted. Movement to characters which are not individually modifiable, such as the individual characters in reserved words, is also inappropriate. Allowing such movement can be frustrating to the user because he can get to locations but is not allowed to modify all of them.

These problems indicate that the unit of movement should be the syntax tree nodes. This is accomplished by providing commands which move the focus from one node to another. The same commands can be used as before, but now they refer to movement within the tree structure. Thus "up" refers to the

BACKGROUND

parent, "down" refers to a child, and "left" and "right" refer to the appropriate sibling. This forces the user to have a better understanding of the language in order to move easily through the structure. Such an in depth understanding of the language is an excellent thing to have, but it is quite a change from current programming practices which tend to emphasize syntactic issues. However with the editor handling all of the syntax problems, the programmer can spend more time understanding the language and what is being done in the program.

This movement between syntax tree elements presents a problem to the conventional point cursor. The cursor should designate the image which represents the entire "editing focus" which is the current program tree node of interest to the user. This is a problem for the syntax directed editor because the focus often contains more than the single character to which the cursor can point. If the cursor points to only the leftmost character of the focus, ambiguities can result. An example is the production for an identifier. The definition of an identifier is :

```
identifier =  
  'AZ|az' {'09|AZ|az'} ;
```

With the focus at the identifier and a point cursor the display would appear as in figure 5-1.

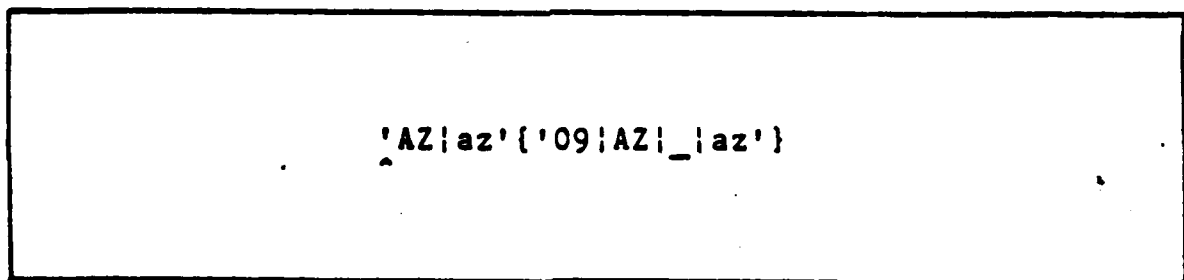


Figure 5-1: Point cursor display

If the focus is moved down to the set 'AZ|az' the display will remain exactly the same because the cursor is already at the leftmost character of the focus.

BACKGROUND

A solution to this problem is an "extended cursor" which is created by highlighting the image of the focus to clearly indicate the extent of what is covered. This highlighting can be in the form of reverse video, color changes, or any similar means of setting off a portion of the screen. Using an enclosing box to signify the extended cursor, figure 5-2 shows the focus at the identifier and then after the focus has moved to the set 'AZ|az'. Most current hardware has some capability for highlighting, but there can be problems with some older devices.

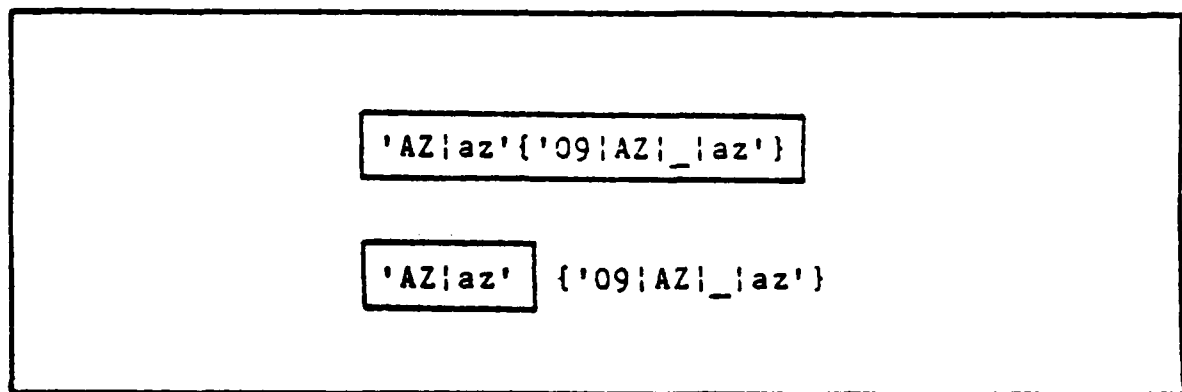


Figure 5-2: Extended cursor display

Movement based on the syntax tree nodes can also provide confusion to a new user because of the way the program is displayed. The display format depends on the format control commands placed in the syntax description (even though these have no effect on the syntax tree itself), while the commands are based on movement within the syntax tree. Thus movement to a right sibling does not necessarily translate to a movement to the right on the screen. Figure 5-3 shows a procedure with the focus at the "decl" node. Invocation of the "right" command moves the focus to the right sibling, in this case the "program_component" node. This results in a cursor movement down to the next line as shown in figure 5-4.

BACKGROUND

```
procedure <identifier>[formal_part] is
    {<decl>}
    {<program_component>}
begin
    <seq_of_stmts>
end [<identifier>];
{<compilation_unit>}
```

Figure 5-3: Focus at "decl" node

```
procedure <identifier>[formal_part] is
    {<decl>}
    {<program_component>}
begin
    <seq_of_stmts>
end [<identifier>];
{<compilation_unit>}
```

Figure 5-4: Display after "right" command,
focus at "program_component"

5.3.2. DISPLAY SELECTION

The physical size of a terminal display screen severely restricts the portion of a large program tree that can be displayed on the screen at any particular time. Determining the portion of the tree to display has been done in different ways by different implementors of syntax directed editors. One simple approach that has been used is to have the user specify exactly which subtree is to be displayed. (Ref 9) The tree is then clipped to fit the viewing screen.

BACKGROUND

Suppression of the display of nodes beyond a certain depth from the subtree root is performed to make additional room on the screen. However this requires an undue amount of work on the part of the user in specifying the subtree and viewing depth. SYNDE uses a modified version of this method using automatic display justification and modular elision. Both of these concepts are discussed in the following paragraphs.

Automatic display justification removes the necessity of the user specifying the subtree to be displayed. The assumption is made that the user is interested in the area surrounding the focus, including the parent and the children. The subtree chosen for display is the one which contains the focus but whose size does not exceed the size of the display screen. This provides the most information for the display with no effort on the part of the user. If the focus itself is too large for the display, it is clipped to show as much as possible.

Modular elision is a modification so the suppression of nodes below a certain depth does not have to be specified by the user. The display of any subtree may be suppressed by command of the user. The image of the "elided" or suppressed subtree is replaced by an arbitrary string to mark the presence of elided material. When inside the elided subtree the editor will consider no view for the screen larger than the elided subtree. When outside the elided subtree, it appears as a mark signifying the presence of material of which the details are unimportant to the higher level.

The elision concept may be thought of as specifying modular levels of information. An elided subtree is displayed only as a unit. Only when one is inside that unit are specific details of any importance. An example is a simple program consisting of two nested procedures. With the focus at the inner procedure the display would look like figure 5-5. Eliding the inner procedure

BACKGROUND

```
procedure outer is
  procedure inner is
    begin
      <seq_of_stmts>
    end inner;
  begin
    <seq_of_stmts>
  end outer;
```

Figure 5-5: Two nested procedures

```
procedure inner is
  begin
    <seq_of_stmts>
  end inner;
```

Figure 5-6: Elided inner procedure

will cause the display to change to that of figure 5-6. Movement further down the tree will allow no tree to be displayed which goes upward past that shown. Moving the focus up to the outer procedure causes the display of the elided material to be suppressed. This gives figure 5-7. The user remains aware of the elided information, but is unconcerned about its content. This is quite similar to the ideas used in modular programming.

BACKGROUND

```
procedure outer is
    +++++
begin
    <seq_of_stmts>
end outer;
```

Figure 5-7: Display of outer procedure
with suppressed inner procedure

5.3.3. DISPLAY IMPLEMENTATION

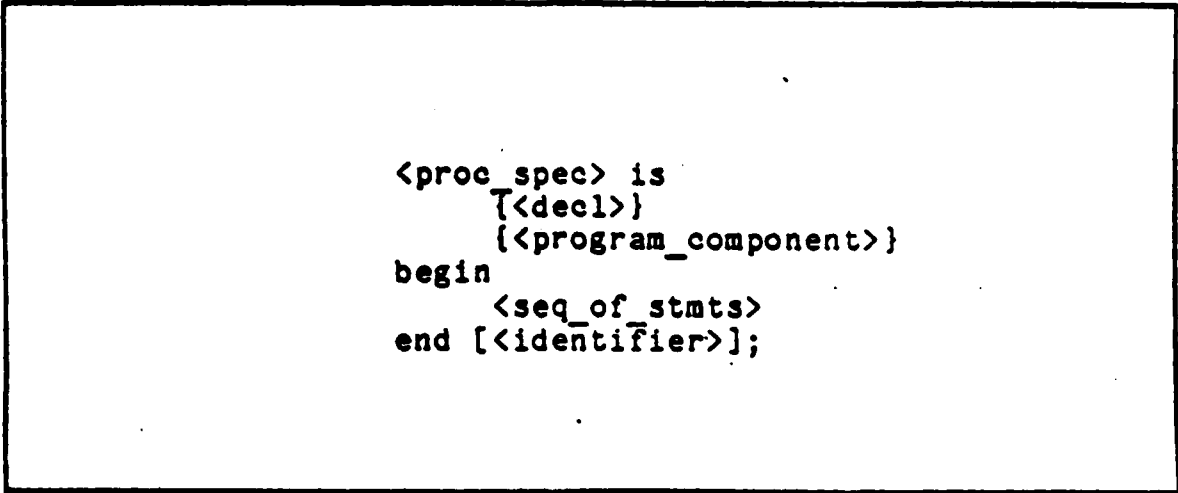
Once the subtree to be displayed has been selected SYNDE begins the process of generating the image based on the syntax tree and the format controls contained in the syntax description. The desired image is generated and stored in an internal data structure representing the current display screen. Changes are made to this structure as the screen is modified. If a new line or character is the same as the old one no change is made to the screen. The actual display routine is discussed below.

The format control characters control the display of the program on the screen. A "^" generates a space, a "@" means to start a new line, and a "+" causes a new line to be generated with an additional level of indentation. To see the results of the various format controls on the generated image one can look at the definition of a `proc_body` :

BACKGROUND

```
proc_body =  
  proc_spec ^ "is"  
    {+ decl}  
    {+ rep_spec ! }  
    {+ program_component}  
  @"begin"  
    + seq_of_stmts  
  [@exceptions !]  
  @"end" [^identifier] ";" ;
```

This would result in the display shown in figure 5-8.



```
<proc_spec> is  
  {<decl>}  
  {<program_component>}  
begin  
  <seq_of_stmts>  
end [<identifier>;
```

Figure 5-8: Display of proc_body

The screen display of the image requires a number of terminal dependent actions such as cursor movement or highlight mode entering or exiting. These are implemented by tying into a screen handling package known as "CURSES"(Ref 1). This package handles the output of the terminal dependent codes for each action. Thus, from the editor's viewpoint, only one type of logical device is dealt with. This frees the user from the worries of proper configuration for the terminal output. All the user has to do is specify the type of terminal he is on to the system sometime before entering the editor, and the device dependent display issues are handled by the power of the computer system.

BACKGROUND

With this background on the syntax tree and how it is displayed, one can look at the individual tools which make up the environment, beginning with the syntax directed editor.

6. SYNDE

The main tool in the environment being developed is the syntax directed editor, known as SYNDE (SYNTAX Directed Editor). By itself it is the most complicated of the current set of tools. In addition SYNDE is the master interface module. It is from the editor that files are created, modified, and listed, and it is the area from which all of the other tools are invoked. SYNDE also has the largest interface to the user and this presents problems of its own. The issue of user friendliness is of major importance for both the input and the output sections of the editor. This chapter will look at how a program tree is developed for the editor, and the changes which have been made to the editor.

6.1. PROGRAM TREE SYNTHESIS

6.1.1. CREATING THE TREE

The syntax tree described in section 5.2 is derived from the language definition. The language syntax definition consists of a set of production rules beginning with the language goal symbol. The definition of this goal symbol describes exactly what is acceptable in the language in terms of other non-terminal and terminal nodes. Thus the goal symbol should be the root of the syntax tree. For the ADA programming language the goal symbol is "compilation" which gives the syntax tree shown in figure 6-1. The syntax description for compilation is:

```

compilation =
  compilation_unit
  {@compilation_unit};

```

The description (in the form needed by the META preprocessor) consists of the name of the element being defined, followed by an equal sign ("="), then the definition of the element, and is terminated by a semicolon (";"). The "@" is

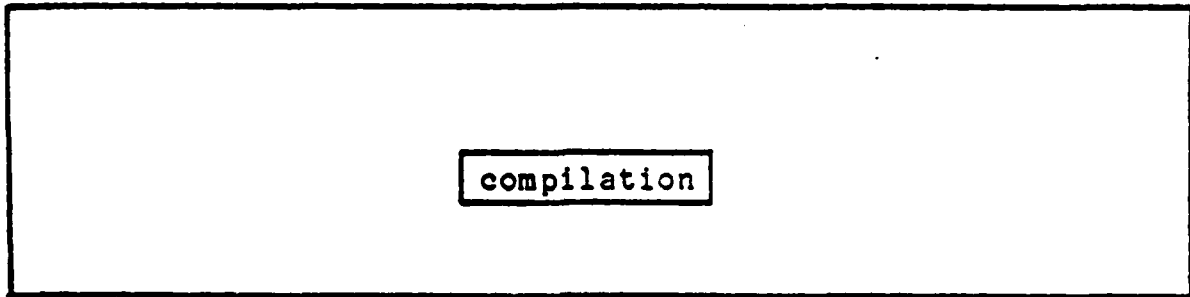


Figure 6-1: ADA syntax tree root

the format control character for a new line discussed in section 5.3.3.

This production is an example of a concatenation production. A new child is created below the node for each of the elements of the definition. It also illustrates the use of a "repeater". The braces ("{" and "}") which surround the second line indicate that the enclosed element may be repeated zero or more times. To become part of the program it must be "established" or selected for inclusion. Unless deleted, it will appear as part of the syntax tree but will not be visible in the output of tools such as the lister.

Once this production is applied to the compilation node, figure 6-1 is transformed into figure 6-2.

Once that production has occurred the children are in turn examined, beginning with the first `compilation_unit`. The syntax definition for a `compilation_unit` is:

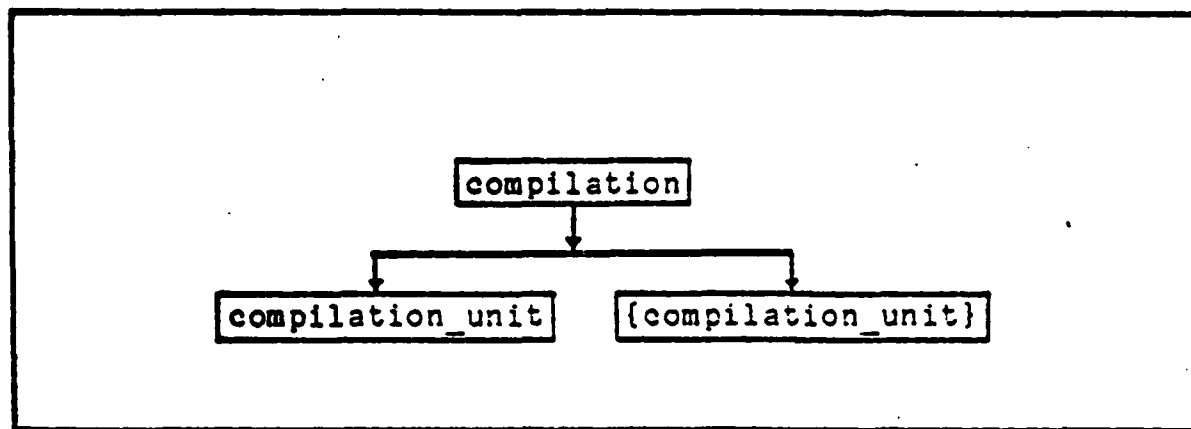


Figure 6-2: Initial expansion of root

```

compilation_unit = <
  proc_body
  func_body
  pack_body
  proc_decl
  func_decl
  pack_decl
  with_use_clause
  subunit
  pragma > ;
  
```

The angle brackets ("**<**" and "**>**") surrounding the list indicate that this is another type of production, an alternation. Each of the alternatives must be a single element, and only one of those elements will be selected as part of the program. The above example illustrates an alternation with nine alternatives. The user selects which of the alternatives he wants synthesized into the tree and that alternative becomes the child of the current node. When the editor help function is turned on, a list of the alternatives available appears at the bottom of the screen to assist the user in making the selection. Selection is currently done by typing enough of the alternative to uniquely identify it (with command completion to assist), however more efficient means such as a light pen or cursor would be a useful extension for systems having such a capability.

Suppose the user selected a `proc_body` as the alternative desired. A child

SYNDE

for the `compilation_unit` node would be generated resulting in the transformation of figure 6-2 to that of figure 6-3.

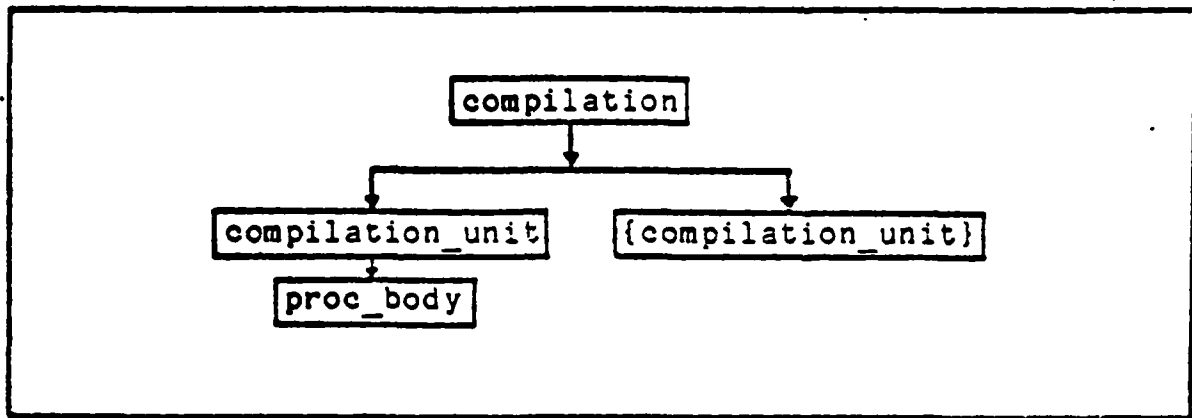


Figure 6-3: Tree after selection of `proc_body`

The synthesis would continue, this time with the `proc_body`. The definition of a `proc_body` is:

```
proc_body =  
  proc_spec ^ "is"  
    {+ decl}  
    {+ rep_spec ! }  
    {+ program_component}  
  @"begin"  
    + seq_of_stmts  
    [ @exceptions ! ]  
  @"end" [ ^identifier ] ";" ;
```

This definition provides the layout of a procedure body. It is a concatenation and introduces a number of new concepts. As before the symbols "@", "+", and "^" are the format control discussed in section 5.3.3 and do not concern the form of the tree itself. The elements enclosed in square brackets ("[" and "]") are optional elements. These are similar to the repeaters discussed earlier, but an option may appear only zero or one time. Options may be synthesized into the tree or removed exactly like repeaters.

Together the options and repeaters are referred to as "conditional"

SYNDE

elements, those whose presence is not required in a valid program. Many, such as the declaration ("decl" in the above syntax definition for a `proc_body`), are often present and should be immediately available for the user to see. Others, such as the exceptions ("exceptions" in the above syntax definition), are rarely used and would be in the way for the average user. It may be desirable to require user action to insert these elements rather than to remove them. This is indicated by the hide indicator ("!"). The hide indicator may be placed at the end of any conditional node to indicate that that element will not automatically be synthesized into the tree. User action is required to make it a visible part of the syntax tree.

The final new item introduced here is the character string enclosed in quotation marks. These strings are the actual terminal symbols which appear in the language. They represent the reserved words or delimiters which are used in the language. They stand for themselves and cannot be further expanded. While necessary for display and listing purposes, such strings appear to provide no useful information to the syntax tree. What a particular string is, and where it belongs in the text output, is easily determined from the language syntax description. These strings are used to determine the extent of various constructs when processing a textual input file. They are not used by the compiler, but are discarded once their purpose as boundary markers has been accomplished. The syntax directed editor has already performed much of the analysis work, and the structure of the tree determines the constructs stored in the tree. Thus the strings add no new information, while occupying storage space. Research has shown that these strings need not occupy space in the tree itself. (Ref 11) Thus to reduce storage requirements these strings are not synthesized into the tree. However they do appear on the user display.

When the above definition of `proc_body` is applied to the `proc_body` node,

figure 6-3 is transformed into figure 6-4. Note that the hidden elements, rep_spec and exceptions, as well as the character strings do not appear in the syntax tree. The display of this tree is shown in figure 6-5.

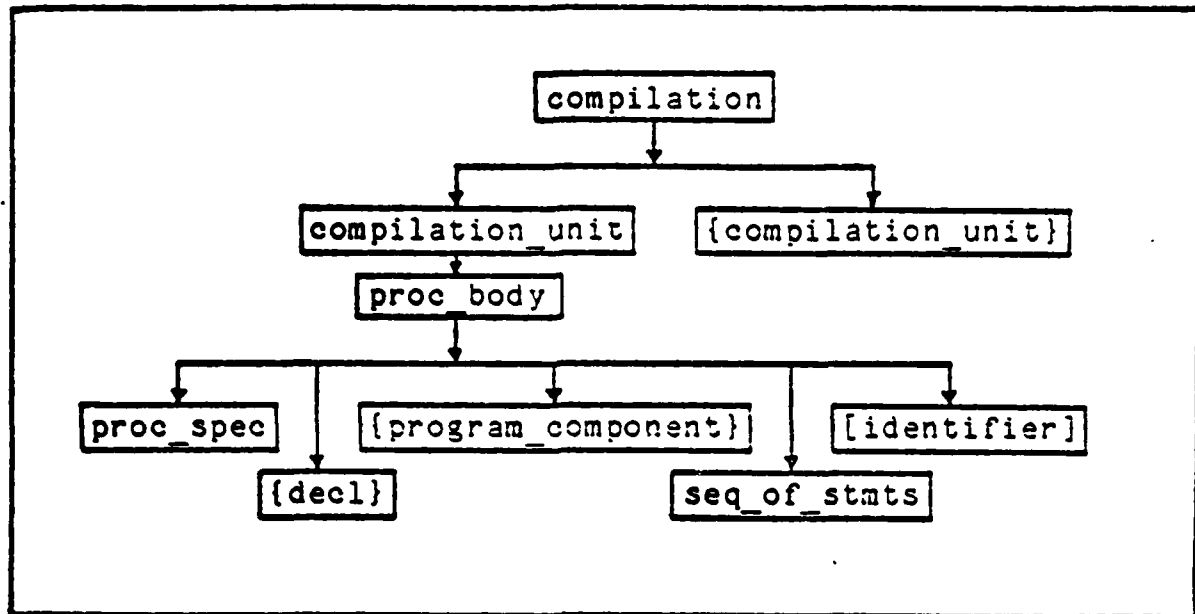


Figure 6-4: Tree after expansion
of proc_body node

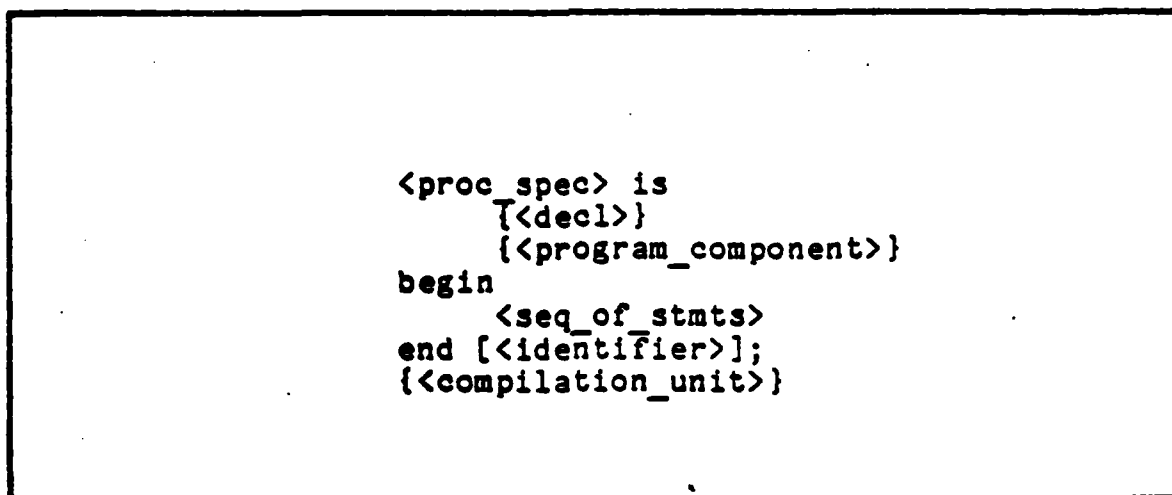


Figure 6-5: Display of tree
shown in figure 6-4

It should be noted that up to this point only one user action has taken place, the selection of a procedure body. The expansion of the syntax tree,

SYNDE

including the placement of required words and delimiters, has taken place automatically. Information indicating the type of nodes which may be synthesized appear for the optional and repeater nodes. This process should continue for those nodes on the "frontier" (the edge of the tree) which are required to be present. The only required nodes not yet examined are `proc_spec` and `seq_of_stmts`. The definition of a `proc_spec` is :

```
proc_spec =  
  "procedure" ^ identifier [formal_part];
```

Once again this is a concatenation and thus may be expanded. Any time that an unconditional node is established whose definition is a concatenation the system should automatically apply the production to get the expanded version.

When this is done for `proc_spec` the subtree of figure 6-6 results.

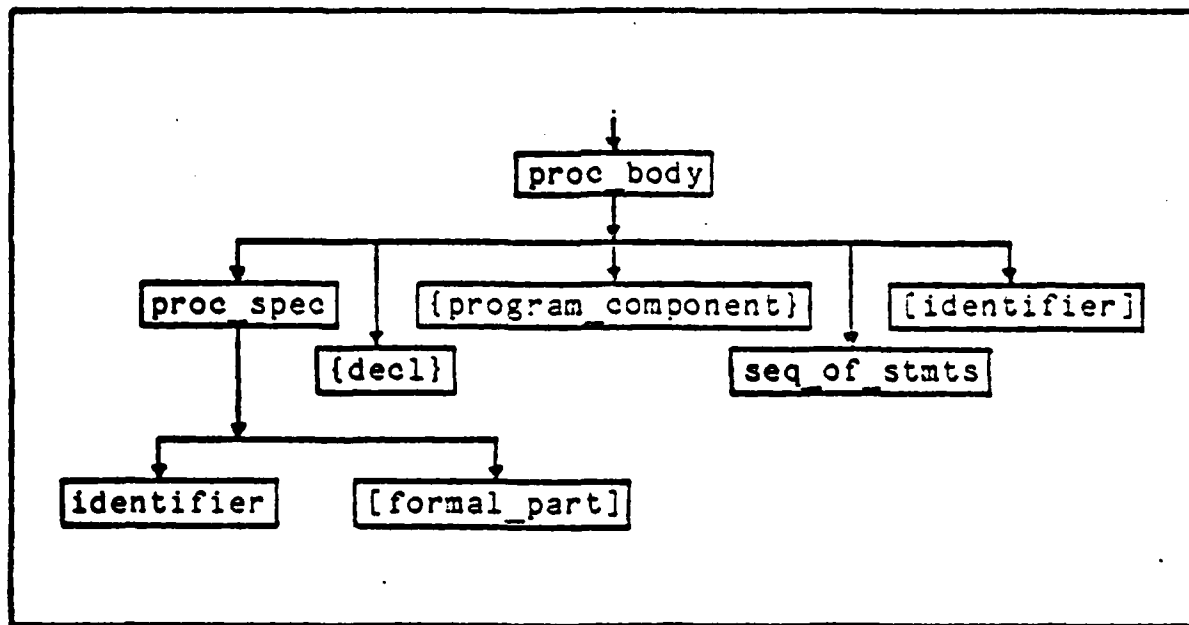


Figure 6-6: Proc_body subtree
after automatic proc_spec expansion

This automatic expansion of unconditional concatenations occurs whenever such a node exists. A look at the definition for an identifier :

SYNDE

```
identifier =
  'AZ|az' {'09|AZ|az'} ;
```

shows that it is also an unconditional concatenation. Thus it is automatically expanded. This continues until no such nodes remain.

The identifier definition shows one of the final types of productions, the set. Sets are compact ways of indicating an alternation which consists of a single character. Elements of a set may be single characters or pairs of characters which specify an inclusive range in the ASCII character set. Thus the set 'AZ|az' is syntactically equivalent to the alternation:

```
letter = <
  "A" "B" "C" "D" "E" "F" "G" "H" "I" "J" "K"
  "L" "M" "N" "O" "P" "Q" "R" "S" "T" "U" "V"
  "W" "X" "Y" "Z" "a" "b" "c" "d" "e" "f" "g"
  "h" "i" "j" "k" "l" "m" "n" "o" "p" "q" "r"
  "s" "t" "u" "v" "w" "x" "y" "z" >;
```

The set notation is more efficient for storage as well as it's obvious ease of use.

Application of the definition to the identifier node yields the subtree shown in figure 6-7. This figure also shows how it would be displayed.

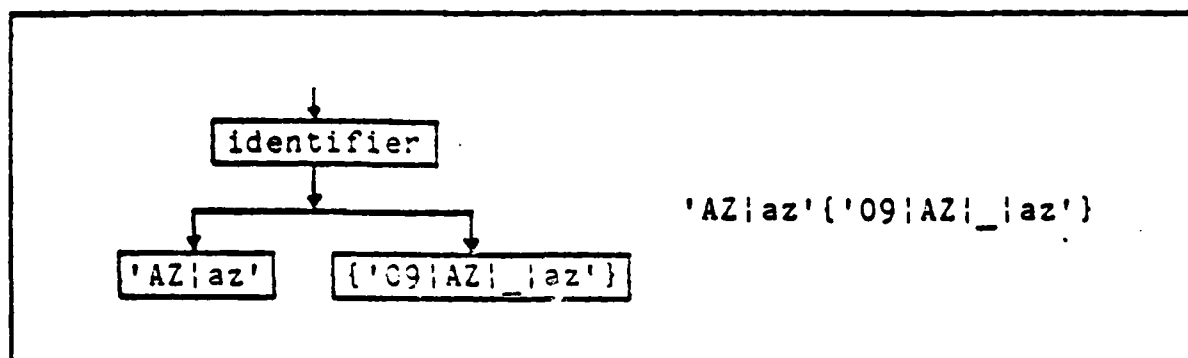


Figure 6-7: Identifier subtree and display

Selection of a set element is done by typing the character desired. The character selected then replaces the name of the set both in the tree and in the display. For example selection of the letter X for the 'AZ|az' set would

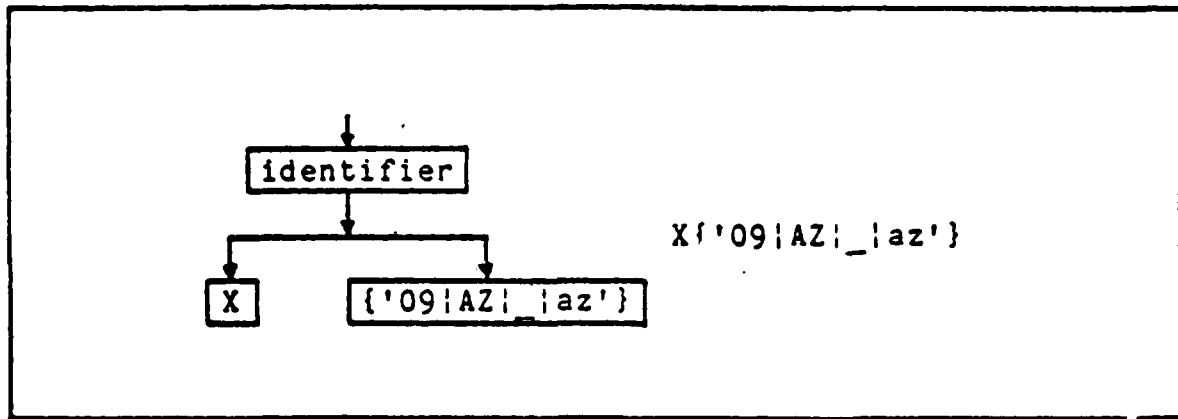


Figure 6-8: Identifier subtree after selection
of first character

transform figure 6-7 into figure 6-8.

6.1.2. MODIFYING THE TREE

The preceding section showed how a tree is originally created and expanded for sets, alternations, and concatenations. At this point what is present are mostly unestablished conditional nodes. These nodes require some user action for them to become part of the tree. For nodes which are defined as a string or concatenation, typing the first character of the displayed name will establish the node. For sets or alternations the selection of one of the alternatives establishes the node. When the node is established it appears without its surrounding brackets or braces. If it is a repeater, another unestablished repeater is inserted immediately to its right. All appropriate expansions of the newly established node are done exactly as discussed above.

The other major means of modifying the tree is through the insertion or deletion of nodes. Both operations take place at the "editing focus". The focus is the current program tree node of interest to the user. The entire subtree below that node is affected by the operations performed.

SYNDE

The function performed by the delete operation depends on the type of node the focus is at. If the node is an option or a repeater the node represents an element which is not needed and the node may simply be deleted. Otherwise the node is required by the parent and may not be removed. However the delete operation for such a required node will clear the node and remove any subtrees which have grown from it, allowing the node to be rebuilt. The only way to eliminate such a node is to delete it's parent.

An example of the use of the delete operation can be seen with the procedure body declaration, previously shown in figure 6-6. If the procedure was to have no formal parameters the focus would be moved to the "formal_part" node, and the delete operation invoked. This would result in the subtree of figure 6-9.

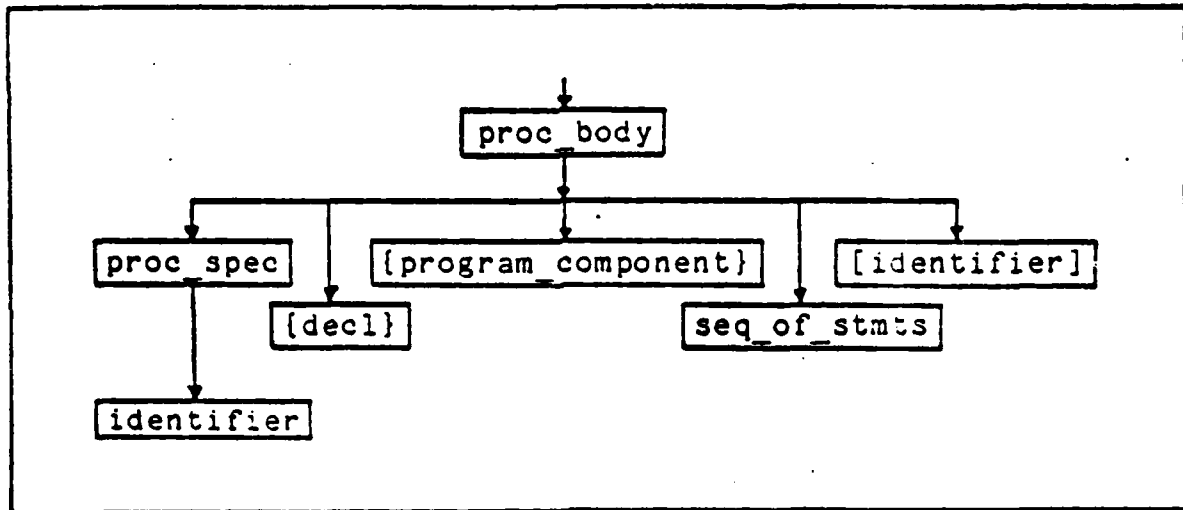


Figure 6-9: Proc_body subtree
after deletion of formal_part

The insert operation may be used to add an option or repeater to the tree. This may be one previously deleted or it may be one which was originally hidden. In either case the focus is moved to a node immediately to the left or

SYNDE

right of where the element is to be inserted. In order to restore the formal_part of the procedure above, the focus would be moved to the identifier node and an "insert right" command issued. Of course there must be something available to insert at that point or an error message will occur. What is available for insertion depends on the syntax definition of the parent of the focus.

"Cutting and pasting" operations are also available by clipping and copying subtrees. More details on those commands are available in the user manual.

6.2. TOOL DEVELOPMENT

The code for SYNDE involved the most changes of any of the tools. The first step was to revise the code to get a working version of the tool on the VAX which was written in standard C. This was a very long step due to the large amount of code written for this tool. A large number of dialect dependent forms and functions were used, requiring extensive rework to obtain a working version.

The code for the display portion of the editor was then modified to take advantage of a display handling package called "curses". "Curses" uses the system terminal description database to provide the proper sequences for items such as entering and exiting reverse video mode. Integration of this package allows a more terminal independent editor environment and frees the user from the worry of entering the proper output command sequences when configuring the environment.

The remaining changes involved the addition of new editor commands. Each new command was added to the case statement controlling command selection, and code for the command was generated. The new commands include the commands to redraw the screen, save current changes to the

SYNDE

program, write the program to another file, and call the editor with a new file to be edited. Details of what each command does can be found in the user manual for the syntax directed editor in Appendix IV.

There are a number of other tools, which, together with the editor discussed in this chapter, form the editor environment. The next chapters look at the other tools in the environment.

META

7. META

"META" is one of the preprocessing tools in the software development environment being developed. It is used to convert a human readable version of a language definition into one which is useful to the other tools such as the editor and lister. A discussion of the syntax description is followed by a discussion of the subset capabilities of META and then a short discussion of the transformation performed by META.

7.1. TOOL DEVELOPMENT

Most of the work for the META preprocessor was done by Captain Ferguson. Code for the tool was transferred to the VAX, and modifications were made to get it running for the larger machine and different dialect. Otherwise the code remained almost intact from the original version.

7.2. META SYNTAX DESCRIPTION

META accepts a form of extended BNF with a few minor changes. (See section 5.1 for a discussion of extended BNF.) The expressions accepted by META must be of a simpler form than those normally found in BNF. For example an option or repeater may only contain one element. Thus

```
... { , identifier }
```

in BNF must be expanded to

```
... { identifiers }  
identifiers =  
    "," identifier;
```

in a META description. Details of these types of restrictions may be found by looking at the description for META found in Appendix I.

The other major difference between META and extended BNF is that META accepts symbols that control the spacing and line display of the elements of the grammar. This is necessary so that the editing and listing programs

META

know how to display the information to the user. These format controls consist of a space mark ("^"), a newline ("@"), and an indentation marker ("+"). Detailed descriptions of what each control does are contained in the META user manual and section 5.3.3. They are all commands that describe how the information is displayed to the user. The inclusion of this type of control in the syntax description results in the use of a standardized format for all programs in the language independent of the user, and relieves the user from the worry of how to format a particular program. While ADA does not require the placement of information in any particular place on a line for syntactic correctness, it is still useful to enforce a standardized format if the computer is handling the formatting. This results in greatly increased readability of code between different programmers or programming teams.

7.3. LANGUAGE SUBSETS

META allows the definition of various subsets of the language. These subsets indicate what elements are to be left out of the language syntax description for that particular subset. Thus the subsets indicated are "excluded" subsets. All elements marked as belonging to subset 0, for example, would be excluded from the syntax description for language subset 0. Alternatives and conditionals may be specified to be non-existent for various subsets.

Language subsetting is a feature which is of special help in learning situations. A language such as ADA is enormous, and can be overwhelming to a programmer attempting to learn the language. With the use of subsets, smaller, more manageable portions of the language set may be used without the details of the other constructs being known. Any program written for this subset would appear exactly as it would in any other subset or in the entire language and would be a valid program in the original language. However the user would be freed from knowing everything about the language right from the

META

beginning. As his knowledge increased larger subsets could be used to introduce new concepts and constructs. The selection of which constructs should be excluded in each subset should be specified by the system manager when the environment is first received and then should be left alone. Individual users may then access the syntax description and available subsets, but should be unable to change it.

META allows up to eight subsets ("0" through "7"). These are specified in the language definition by placing a "\$" followed by a series of digits after the conditional or alternative to be left out for that set. When a subset is specified to META, those entries marked with that subset number are not included. Any grammar rules which are not accessible because of the subset indication are not included in the language definition.

7.4. META CONVERSIONS

The information in human readable form is not the most efficient way for the other tools to access the required information, so META converts it to another form. It first processes the syntax description to insure that there are no syntax errors in the description. Any undefined items, multiple definitions of items, illegal formats, or other errors are flagged and the program terminates.

Ways of increasing the efficiency and readability of the syntax definition by combining production rules are noted but are not carried out. This includes noting the location of single alternatives and single conditional elements. These types of elements may be combined into other elements in the description to produce a more compact, and more readable, form of the syntax description. It will also reduce the number of decisions required by the user when entering a program. Although the syntax directed editor attempts to perform these optimizations while it is running, if the syntax description is to be used often it

META

should be rewritten to take advantage of the optimizations suggested by META.

A version of the description in a form that is usable by the other tools is built using "syntax nodes". These nodes contain a flag detailing the type of information contained in the node plus a pointer to the next element or a string that represents that element. These are more easily accessible to the other tools, and the conversion done here eliminates the need for repeating these lengthy conversions later by storing the description in the form needed by the other tools. The final output of META is the converted syntax description stored in a file (name.sdf) and an output summary detailing what information was processed. Only the ".sdf" file is used by the other tools.

OTHER TOOLS

8. OTHER TOOLS

In addition to the editor (SYNDE) and META, two other tools are part of the current syntax directed editor environment. These are Config and Lister, and will be discussed in this chapter.

8.1. CONFIG

The Config program (short for configuration) is another one of the preprocessing tools in the environment being developed. The purpose of this tool is to allow configuration of the editor to correspond to a particular user and a particular terminal. The user is able to enter the sequences of keystrokes which are used to invoke each of the commands in the editor. (See the user manual for SYNDE in appendix IV for details of the available editor commands.) These sequences can be any sequence of up to seven keystrokes beginning with a non-printable character. Each user can have a personalized set of command calls, even though the set of commands remains constant. Thus each individual can fine tune the input commands to ones with which he or she feels comfortable. This is an effort to increase the user friendliness of the overall environment and it does not reduce the abilities of the particular tools. Responses are stored in a file titled "terminal.tdf" which is used by the editor to obtain the needed input and output command strings.

8.1.1. TOOL DEVELOPMENT

The code for a version of Config which was designed for use on a microcomputer was transferred to the VAX. Work was done to get that version working for the larger system, and then modifications were made. The second version of the program allowed the user to have Config obtain the proper output command sequences from the host system's terminal description database and enter them in the "terminal.tdf" file. Modifications to other tools have added packages which automatically retrieve this information from the system

OTHER TOOLS

database, and thus the ability to copy this information from the system database into the "terminal.tdf" file became obsolete. Further versions of the tool were necessary as new commands were added to the editor. Each new command involved a simple change to Config to allow the entering of the new input command sequence.

8.1.2. GENERAL INFORMATION

There are two major categories of information covered by Config : input commands, and display or output control command sequences. These will be discussed in some detail after discussion of information which pertains to the entire Config program.

Config travels through a series of loops which allow the user to selectively change the input commands or output sequences as often as necessary. After a prompt the user is required to enter a response. A "Y" or "y" is treated as a "yes" response, while a carriage return or any other key is treated as a "no". If possible, a negative response leaves the information unchanged. As the commands are entered they are echoed to the terminal so that the user can see what has been entered. More details on the form of the command echoing is available in the user manual.

8.1.3. INPUT COMMAND SEQUENCES

The configuration of the input commands allows the users to specify command sequences with which they are comfortable. Each user of the editor can have a personalized set of command sequences. All commands should start with a non-printable character because these are not used in the source code which will be entered through the editor. Each command must be entered and must be unique.

OTHER TOOLS

8.1.4. OUTPUT COMMAND SEQUENCES

The output command section is of less importance than the input commands but can not be ignored. This is due to expansions which have occurred in the environment since it was originally designed. The majority of the sequences entered in the output section are no longer used by the environment unless the terminal is of an unknown or uncommon type whose capabilities are not known to the operating system. If the terminal type is unknown, the editor environment will use the sequences entered through Config. If the type is known, the environment will automatically use the information contained in the system's terminal capability database. Thus the majority of the information in the output section is of no concern to the average user.

Two major types of information are entered in the output section. Three of the screen display items, the window divider, the number of spaces used for a tab, and an elision marker, have no predefined values and therefore must be entered by the user. The rest of the commands are the terminal dependent features. These need not be entered by the user and are only used if the operating system does not have the required information. These command sequences are appropriately marked in the prompts to the user in the Config program.

More details on what commands are entered and how they are displayed can be found in the user manuals for the editor (SYNDE) and for the Config program.

8.2. LISTER

A tool to generate a text listing of a program is needed because of the form that a program is stored in by the editor. The syntax tree description is what is stored, unlike current practices which store the text file. Thus the

OTHER TOOLS

program is stored in a state halfway between conventional source programs and conventional compiled versions of those programs. Conversion from this intermediate form is needed both for human readability and to allow the program to be submitted to other more conventional tools.

"Lister", the source program listing tool, may be called from the editor or may be called on it's own. Input for Lister is a syntax tree such as the one generated by the editor. The output is a human readable text listing of the program similar to the type of source code generated by most programmers before the development of syntax directed editors. The format of the generated listing is controlled by the format control characters imbedded in the syntax description.

The code for Lister required few changes from the original version. As with all of the tools, when the code was transferred to the VAX changes needed to be made to make the code compatible with the standard dialect and new system. These were the only changes needed for Lister.

As mentioned, Lister may be called in one of two manners. When called by itself it generates the requested listing based on the argument list, then returns to the operating system. When called from the editor it generates a listing of the file currently being edited, then returns to the editor. The job performed is the same in either case, only the return location is different.

The listing generated contains only those elements which have been synthesized into the program, or those which are required to be present. Optional or repeated elements which have not been selected will not appear in the listing, even though they may be visible in the editor. Required elements which have not been expanded will appear as the character string name of the item to be entered surrounded by angle brackets ("<" and ">"). This will cause

OTHER TOOLS

an error if an attempt is made to compile the program, but an error would occur at that point in any case. This way the user can see what type of element needs to be inserted at that point. Once again the user friendliness issues have been considered.

That concludes the discussion of the individual tools which are part of the current syntax directed editor environment. The final chapter presents some conclusions and recommendation about the environment which contains these tools.

CONCLUSIONS AND RECOMMENDATIONS

9. CONCLUSIONS AND RECOMMENDATIONS

9.1. CONCLUSIONS

A syntax directed editor appears to be a very powerful tool for the software development environment of the future. This type of editor frees the user of the worries of syntactic errors and allows the user to concentrate on other areas such as a better understanding of the language and more efficient design of programs. The power of the computer will begin to be used in an area which it has long been ignored : the development of computer software. Although a knowledge of the language syntax will still be important, the mistake prone step of guaranteeing correct syntax will be shifted from the user to the computer.

The environment developed in this research effort is not an ultimate environment. It is merely another step in that direction. Additions and modifications have been made so that this environment is both more powerful and of wider use than its predecessor.

This environment, as was its predecessor, was developed for use with the ADA programming language. However it is not restricted to any particular language. The editor, lister, configuration preprocessor, and language preprocessor, are all language independent and require only a new language syntax description to be used with other languages. The interpreter and compiler are language dependent and would require a fair amount of work to develop them for other languages. However syntactically correct programs can be obtained from the editor and lister to send to conventional tools.

Finally the use of such a software development environment is especially useful in an academic or other learning environment. Use of subsets of the language allows the introduction of the language in smaller, more controlled

CONCLUSIONS AND RECOMMENDATIONS

units. A syntax directed editor allows concentration on language structure and program design while removing the "bookkeeping chores" that deal with correct syntax. Such tools should be of great assistance to the upcoming generation of computer programmers.

9.2. RECOMMENDATIONS

The recommendations for continued work in this area fall into five major categories. These are conversion to ADA, incremental compilation, multi-tasking, program documentation, and user interface improvements.

9.2.1. CONVERSION OF CODE TO ADA

The current direction in the Department of Defense is for future software to be developed in the ADA language. Many of the features of ADA, such as tasking, will allow further developments in the current environment package to be done more easily. In addition it will provide a portable package which will be of use throughout the Department of Defense. All of the code for the syntax directed editor environment is currently written in the programming language C. This code would need to be converted to ADA before these benefits could be realized. Although conversion of such a large amount of C code will be a major effort, it will result in a product which should provide benefits in many applications over a long period of time.

9.2.2. INCREMENTAL COMPILATION

Changes in a piece of code in a file requires that the entire file be recompiled even though only a small portion of the code was affected. This happens because the compiler does not know where the changes were made and how much of the syntax tree they affect. Some modifications have been made to the current environment to begin the work needed for incremental compilation. These involve modifications of the nodes of the syntax tree to

CONCLUSIONS AND RECOMMENDATIONS

note whether code has been generated for the subtree, or if source code changes have occurred in the subtrees for which code had previously been generated. If changes have been made, the code for that subtree must be regenerated. This is only a simple first step towards the use of incremental compilation. Investigation into the requirements of such compilation needs to be done and integrated into the current environment. Successful results at other institutes such as Carnegie -Mellon University (Ref 7,10) would provide a useful starting point for such efforts.

9.2.3. MULTI-TASKING

In almost all systems the computer spends a lot of the time idle and waiting for someone to do something. In addition, much of the work done by such tools as the compiler can be accomplished without the user sitting there staring at the screen. An ability to initiate something like an incremental compilation in parallel would reduce the amount of time the user wastes waiting for the computer to finish its tasks. Such work could be performed during the unused cycles while the user is editing other sections of the code. Thus the code generation for most of the other procedures or routines could be completed by the time the user finishes typing in the last procedure. This would substantially reduce the time required after the user completes the editing process to produce the code. Other areas which would benefit from a parallel task can certainly be found. This ability should be integrated into the environment both for efficiency and especially for increased user friendliness.

9.2.4. PROGRAM DOCUMENTATION

Proper and complete documentation of the code written by a programmer is an important part of the development of useful computer software. Although comments within the code are not part of the syntax description of the language and thus are ignored by the compiler, they provide important documentation for

CONCLUSIONS AND RECOMMENDATIONS

the code. The current editor environment has not addressed the issue of commenting, and has no means of inserting comments in the source code generated by the editor. A means of adding, or possibly requiring, program code documentation needs to be developed for the editor environment.

A possible solution to this problem would be the addition of a comment field to each node within the syntax tree. This field could contain a character string corresponding to a comment, and could be suppressed during most of the editing session or suppressed until a text file of the program is listed. There are a number of problems with this approach, including the limiting of the comment placement. It is mentioned here only as a possible starting place for much needed future research.

9.2.5. USER INTERFACE IMPROVEMENTS

The software development environment which has been developed was designed with a user friendly interface in mind. The user interface obtained was sufficiently friendly for the designers of the environment. However, neither this software development environment nor its prototype have had the opportunity to be widely utilized. Additional commands have been added to improve user capabilities but this was based only on the designer's incomplete knowledge of the users. The environment should be tested with a wide range of users and their comments and criticisms obtained. Addition of new commands, deletion of unused commands, or modifications to the current set should be done to improve the friendliness of the environment and make it more responsive to the users. Although no system can satisfy all users, a friendly, useful environment will be a boost to productivity.

Computers have changed the way the world operates over the last few

CONCLUSIONS AND RECOMMENDATIONS

years. Finally the power of the computer is beginning to return and provide help to the way that computer software is developed.

BIBLIOGRAPHY

1. Arnold, Kenneth C. Screen Updating and Cursor Movement Optimization: A Library Package. Computer Science Division, University of California, Berkeley.
2. Barnes, J.G.P. Programming in ADA. Addison-Wesley 1982.
3. Department of Defense. Military Standard ADA programming Language. Washington, D.C. January 1983, (ANSI/MIL-STD-1815A).
4. Department of Defense. Reference Manual for the ADA Programming Language. Washington, D.C. July 1980, (AD-A090709)
5. Department of Defense. Requirements for ADA Programming Support Environments - Stoneman. Washington, D.C. 1980.
6. Feiler, Peter H. and Raul Medina-Mora. An Incremental Programming Environment. Carnegie -Mellon University, Pittsburgh, PA. Department of Computer Science, April 1980.
7. Feiler, Peter H. A Language-Oriented Interactive Programming Environment Based on Compilation Technology. PhD Thesis Carnegie - Mellon University. 1982
8. Ferguson, Scott E. A Syntax-Directed Programming Environment for the ADA Programming Language. Master's Thesis Air Force Institute of Technology. December 1982. (AD-A053032).
9. MacLennan, Bruce J. The Automatic Generation of Syntax-Directed Editors. Naval Postgraduate School, Monterey, CA., 1981.
10. Medina-Mora, Raul and David S. Notkin. ALOE Users' and Implementors' Guide. Technical Report CMU-CS-81-145, Carnegie - Mellon University Department of Computer Science. November 1981.
11. Shockley, William R. and Daniel P. Haddow. A Conceptual Framework for Grammar Driven Synthesis. Master's Thesis, Naval Postgraduate School, Monterey, CA., 1981.

META SYNTAX DESCRIPTION LANGUAGE

APPENDIX I META SYNTAX DESCRIPTION LANGUAGE

This appendix contains a definition of the META syntax description language. The description is given in META.

```
syntax =  
    rule  
    { rule } ;  
  
rule =  
    identifier ^ "="  
        + definition ";" ;  
  
identifier =  
    'AZ|az' { '09|AZ|_|az' } ;  
  
definition = <  
    alternation  
    concatenation > ;  
  
alternation =  
    "<" ^ element { ^ element } ^ ">" ;  
  
concatenation =  
    term { ^ term } ;  
  
element =  
    primary [ ^ "!" ] [ ^ index ] ;  
  
term = <  
    option  
    repeater  
    primary > ;  
  
primary =  
    [ '+' | '^' ^ ] factor [ ^ ' ' ] ;  
  
index =  
    "$" { '07' } ;  
  
option =  
    "[" element "]" ;  
  
repeater =  
    "{" element "}" ;
```

META SYNTAX DESCRIPTION LANGUAGE

```
factor = <
    identifier
    string
    set > ;

string =
    " "" { , ~ , } "" " ;

set =
    " " pair { pairs } " " ;

pair =
    , ~ , [ , ~ , ] ;

pairs =
    " | " pair ;
```

APPENDIX II

META DESCRIPTION FOR ADA

This appendix contains the META description for the ADA programming language. It is adapted from the 1980 ADA reference manual.

```

compilation =
    compilation_unit
    { @ compilation_unit $0 } ;

```

```

compilation_unit = <
    proc_body
    func_body $0
    pack_body $0
    proc_decl $0
    func_decl $0
    pack_decl $0
    with_use_clause $0
    subunit $0
    pragma $0 > ;

```

```

proc_body =
    proc_spec ^ "is"
        { + decl }
        { + rep_spec ! $0 }
        { + program_component }
    @ "begin"
        + seq_of_stmts
    [ @ exceptions ! $0 ]
    @ "end" [ ^ identifier ] ";" ;

```

```

func_body =
    func_spec ^ "is"
        { + decl }
        { + rep_spec ! }
        { + program_component }
    @ "begin"
        + seq_of_stmts
    [ exceptions ! ]
    @ "end" [ ^ designator ] ";" ;

```

META DESCRIPTION FOR ADA

```
pack_body =
    "package" ^ "body" ^ identifier ^ "is"
        { + decl }
        { + rep_spec ! }
        { + program_component }
    [ @ body_part ]
    @ "end" [ ^ identifier ] ";" ;

proc_decl = <
    proc_spec_semi
    generic_proc_decl
    generic_proc_instant > ;

func_decl = <
    func_spec_semi
    generic_func_decl
    generic_func_instant > ;

pack_decl = <
    pack_spec
    generic_pack_decl
    generic_pack_instant > ;

with_use_clause =
    with_clause [ ^ use_clause ] ;

subunit =
    "separate" ^ "(" name ")" ^ subunit_body ;

pragma =
    "pragma" ^ identifier [ actual_param_part ] ";" ;

proc_spec =
    "procedure" ^ identifier [ formal_part $0 ] ;

program_component = <
    proc_body
    func_body $0
    pack_body $0
    task_body $0
    proc_stub $0
    func_stub $0
    pack_stub $0
    task_stub $0
    pack_decl $0
    task_decl $0 > ;
```


META DESCRIPTION FOR ADA

```
decl = <
    object_decl
    type_decl $0
    subtype_decl $0
    number_decl $0
    func_decl $0
    proc_decl $0
    pack_decl $0
    task_decl $0
    exception_decl $0
    rename_object $0
    rename_exception $0
    rename_proc $0
    rename_func $0
    rename_pack $0
    rename_task $0
    use_clause $0
    pragma $0 > ;
```

```
rep_spec = <
    length_spec
    enum_type_rep
    record_type_rep
    address_spec > ;
```

```
seq_of_stmts =
    stmt
    { @ stmt } ;
```

```
exceptions =
    "exception"
    { + exception_handler } ;
```

```
identifier =
    'AZ|az' { '09|AZ|az' } ;
```

```
func_spec =
    "function" ^ designator [ ^ formal_part ]
    ^ "return" ^ subtype_indication ;
```

```
designator = <
    identifier
    operator_symbol > ;
```

META DESCRIPTION FOR ADA

```
body_part =
    "begin"
    + seq_of_stmts
    [ @ exceptions ! ] ;

proc_spec_semi =
    proc_spec ";" ;

generic_proc_decl =
    "generic"
    { + generic_formal_param }
    @ proc_spec ";" ;

generic_proc_instant =
    "procedure" ^ identifier ^ "is"
    ^ generic_instant ";" ;

func_spec_semi =
    func_spec ";" ;

generic_func_decl =
    "generic"
    { + generic_formal_param }
    @ func_spec ";" ;

generic_func_instant =
    "function" ^ designator ^ "is"
    ^ generic_instant ";" ;

pack_spec =
    "package" ^ identifier ^ "is"
    { + decl }
    [ @ private_part ]
    @ "end" [ ^ identifier ] ";" ;

generic_pack_decl =
    "generic"
    { + generic_formal_param }
    @ pack_spec ;

generic_pack_instant =
    "package" ^ identifier ^ "is"
    ^ generic_instant ";" ;

with_clause =
    "with" ^ name { names } ";" ;
```

META DESCRIPTION FOR ADA

```
use_clause =
    "use" ^ name { names } ";" ;

name = <
    identifier
    index_component $0
    selected_component $0
    slice $0
    attribute $0
    func_call $0
    operator_symbol $0 > ;

subunit_body =
    proc_body
    func_body
    pack_body
    task_body > ;

actual_param_part =
    "(" param_assoc { param_assocs } ")" ;

formal_part =
    "(" param_decl { param_decls } ")" ;

object_decl =
    id_list ":" [ ^ "constant" ] ^ object_type
    [ ^ initial ] ";" ;

type_decl =
    "type" ^ identifier [ ^ discrim_part ! ]
    [ ^ type_body ] ";" ;

subtype_decl =
    "subtype" ^ identifier ^ "is" ^
    subtype_indication ";" ;

number_decl =
    id_list ":" ^ "constant" ^ initial ";" ;

task_decl =
    "task" [ ^ "type" ] ^ identifier
    [ ^ task_def ] ";" ;

exception_decl =
    id_list ":" ^ "exception" ";" ;
```

META DESCRIPTION FOR ADA

```
rename_object =
    identifier ":" ^ name "renames" ^ name ";" ;

rename_exception =
    identifier ":" ^ "exception"
        ^ "renames" ^ name ";" ;

rename_proc =
    proc_spec ^ "renames" ^ name ";" ;

rename_func =
    func_spec ^ "renames" ^ name ";" ;

rename_pack =
    "package" ^ identifier ^ "renames" name ";" ;

rename_task =
    "task" ^ identifier ^ "renames" ^ name ";" ;

length_spec =
    "for" ^ attribute ^ "use" ^ expression ";" ;

enum_type_rep =
    "for" ^ name ^ "use" ^ aggregate ";" ;

record_type_rep =
    "for" ^ name ^ "use"
        + record_rep ;

address_spec =
    "for" ^ name ^ "use" ^ "at" ^ simple_exp ";" ;

task_body =
    "task" ^ "body" ^ identifier ^ "is"
        { + decl }
        { + rep_spec ! }
        { + program_component }
    @ "begin"
        + seq_of_stmts
    [ @ exceptions ! ]
    @ "end" [ ^ identifier ] ";" ;

proc_stub =
    proc_spec ^ "is" ^ "separate" ";" ;

func_stub =
    func_spec ^ "is" ^ "separate" ";" ;
```

META DESCRIPTION FOR ADA

```
pack_stub =
    "package" ^ "body" ^ identifier ^ "is"
                                "separate" ";" ;

task_stub =
    "task" ^ "body" ^ identifier ^ "is" ^
                                "separate" ";" ;

stmt =
    { label ^ ! $0 } simple_stmt ;

exception_handler =
    "when" ^ exception_choice { exception_choices } ^
                                "=>"
    + seq_of_stmts ;

subtype_indication =
    name [ ^ constraint $0 ] ;

operator_symbol =
    char_string ;

generic_formal_param = <
    param_decl_semi
    generic_proc
    generic_func
    generic_type > ;

generic_instant =
    "new" ^ name [ generic_assocs ] ;

private_part =
    "private"
        { + decl }
        { + rep_spec ! } ;

names =
    "," ^ name ;

indexed_component =
    name "(" expression { expressions } ")" ;

selected_component =
    name "." component ;
```

META DESCRIPTION FOR ADA

```
slice =
    name "(" discrete_range ")" ;

attribute =
    name "'" identifier ;

func_call =
    name [ actual_param_part ] ;

param_assoc =
    [ param_link ^ ] expression ;

param_assocs =
    "," ^ param_assoc ;

param_decl =
    id_list ":" [ ^ "in" ] [ ^ "out" ]
        ^ subtype_indication [ ^ initial ] ;

param_decls =
    ";" ^ param_decl ;

id_list =
    identifier { identifiers } ;

object_type = <
    subtype_indication
    array_type_def $0 > ;

initial =
    "!=" ^ expression ;

discrim_part =
    "(" discrim_decl { discrim_decls } ")" ;

type_body =
    "is" ^ type_def ;

task_def =
    "is"
        { + entry_decl }
        { + rep_spec ! }
    @ "end" [ ^ identifier ] ;

aggregate =
    "(" component_assoc { component_assocs } ")" ;
```

META DESCRIPTION FOR ADA

```
expression = <
    relation
    and_comp
    or_comp
    and_then_comp $0
    or_else_comp $0
    xor_comp $0 > ;

record_rep =
    "record" [ ^ align_clause ]
                { + name_location }
    @ "end" ^ "record" ";" ;

simple_exp =
    [ unary_operator ! ] term { terms } ;

label =
    "<<" identifier ">>" ;

simple_stmt = <
    assignment_stmt
    if_stmt
    loop_stmt
    proc_call
    case_stmt $0
    block $0
    exit_stmt $0
    return_stmt $0
    goto_stmt $0
    entry_call $0
    delay_stmt $0
    abort_stmt $0
    raise_stmt $0
    code_stmt $0
    accept_stmt $0
    selective_wait $0
    cond_entry_call $0
    timed_entry_call $0
    null_stmt $0 > ;

exception_choice = <
    name
    "others" > ;

exception_choices =
    "|" ^ exception_choice ;
```

META DESCRIPTION FOR ADA

```
constraint = <
    range_constraint
    float_pt_constraint
    fixed_pt_constraint
    index_constraint
    discrim_constraint > ;

char_string =
    "" { , ~ } "" ;

param_decl_semi =
    param_decl ";" ;

generic_proc =
    "with" ^ proc_spec [ ^ generic_is ] ";" ;

generic_func =
    "with" ^ func_spec [ ^ generic_is ] ";" ;

generic_type =
    "type" ^ identifier [ ^ discrim_part ] ^ "is"
    ^ generic_type_def ";" ;

generic_assocs =
    "(" generic_assoc { generic_assoc } ")" ;

expressions =
    "," ^ expression ;

component = <
    identifier
    "all"
    operator_symbol > ;

discrete_range = <
    type_range
    range > ;

param_link =
    identifier ^ "=>" ;

identifiers =
    "," ^ identifier ;

discrim_decl =
    id_list ":" ^ subtype_indication [ ^ initial ] ;
```


META DESCRIPTION FOR ADA

```
discrim_decls =  
    ";" ^ discrim_decl ;  
  
array_type_def = <  
    constrained_array  
    unconstrained_array > ;  
  
type_def = <  
    range_constraint  
    float_pt_constraint  
    fixed_pt_constraint  
    array_type_def  
    record_type_def  
    enum_type_def  
    access_type_def  
    derived_type_def  
    private_type_def > ;  
  
entry_decl =  
    "entry" ^ identifier [ ^ entry_dimension ]  
                        [ ^ formal_part ] ";" ;  
  
relation =  
    simple_exp [ ^ relation_part ! ] ;  
  
and_comp =  
    relation ^ { and_relation } ;  
  
or_comp =  
    relation ^ { or_relation } ;  
  
and_then_comp =  
    relation ^ { and_then_relation } ;  
  
or_else_comp =  
    relation ^ { or_else_relation } ;  
  
xor_comp =  
    relation ^ { xor_relation } ;  
  
component_assoc =  
    [ choice_link ^ ] expression ;  
  
component_assocs =  
    "," ^ component_assoc ;
```

META DESCRIPTION FOR ADA

```
align_clause =
    "at" ^ "mod" ^ simple_exp ";" ;

name_location =
    name ^ "at" ^ simple_exp ^ "range" ^ range ";" ;

unary_operator = <
    "+"
    "-"
    "not" > ;

term =
    factor { factors } ;

terms =
    add_op term ;

assignment_stmt =
    name ^ " := " ^ expression ";" ;

if_stmt =
    "if" ^ expression ^ "then"
        + seq_of_stmts
    { @ elsif_part }
    [ @ else_part ]
    @ "end" ^ "if" ";" ;

loop_stmt =
    [ tag ^ ! $0 ] [ iteration_clause ^ ] "loop"
        + seq_of_stmts
    @ "end" ^ "loop" [ ^ identifier ! $0 ] ";" ;

proc_call =
    name [ actual_param_part $0 ] ";" ;

case_stmt =
    "case" ^ expression ^ "is"
        { + cases }
    @ "end" ^ "case" ";" ;

block =
    [ tag ^ ! ] [ declare ]
    @ "begin"
        + seq_of_stmts
    [ @ exceptions ! ]
    @ "end" [ ^ identifier ! ] ";" ;
```

META DESCRIPTION FOR ADA

```
exit_stmt =
    "exit" [ ^ name ] [ ^ when_clause ] ";" ;

return_stmt =
    "return" [ ^ expression ] ";" ;

goto_stmt =
    "goto" ^ name ";" ;

entry_call =
    name [ actual_param_part ] ";" ;

delay_stmt =
    "delay" ^ simple_exp ";" ;

abort_stmt =
    "abort" ^ name { names } ";" ;

raise_stmt =
    "raise" [ ^ name ] ";" ;

code_stmt =
    qualified_exp ";" ;

accept_stmt =
    "accept" ^ name [ formal_part ]
        [ ^ accept_action ] ";" ;

selective_wait =
    "select" [ condition_link ]
        + select_alternative
    { @ or_clause }
    [ @ else_part ]
    @ "end" ^ "select" ";" ;

cond_entry_call =
    "select"
        + entry_call
        [ + seq_of_stmts ]
    @ "else"
        + seq_of_stmts
    @ "end" ^ "select" ";" ;

null_stmt =
    "null" ";" ;
```

META DESCRIPTION FOR ADA

```
timed_entry_call =
    "select"
        + entry_call
        [ + seq_of_stmts ]
    @ "or"
        + delay_alternative
    @ "end" ^ "select" ";" ;

range_constraint =
    "range" ^ range ;

float_pt_constraint =
    "digits" ^ simple_exp [ ^ range_constraint ] ;

fixed_pt_constraint =
    "delta" ^ simple_exp [ ^ range_constraint ] ;

index_constraint =
    "(" discrete_range { discrete_ranges } ")" ;

discrim_constraint =
    "(" discrim_spec { discrim_specs } ")" ;

generic_is =
    "is" ^ generic_name ;

generic_type_def = <
    generic_discrete
    generic_integer
    generic_float
    generic_fixed
    array_type_def
    access_type_def
    private_type_def > ;

generic_assoc =
    [ param_link ^ ] generic_actual_param ;

type_range =
    name [ ^ range_constraint ] ;

range =
    simple_exp ".." simple_exp ;
```

META DESCRIPTION FOR ADA

```
constrained_array =  
    "array" ^ index_constraint ^ "of"  
                                ^ subtype_indication ;  
  
unconstrained_array =  
    "array" ^ "(" index { indices } ")" ^ "of"  
                                ^ subtype_indication ;  
  
record_type_def =  
    "record"  
        + component_list  
    @ "end" ^ "record" ;  
  
enum_type_def =  
    "(" enum_lit { enum_lits } ")" ;  
  
access_type_def =  
    "access" ^ subtype_indication ;  
  
derived_type_def =  
    "new" ^ subtype_indication ;  
  
private_type_def =  
    [ "limited" ^ ] "private" ;  
  
entry_dimension =  
    "(" discrete_range ")" ;  
  
relation_part = <  
    relational  
    in_range $0 > ;  
  
and_relation =  
    "and" ^ relation ;  
  
or_relation =  
    "or" ^ relation ;  
  
and_then_relation =  
    "and" ^ "then" ^ relation ;  
  
or_else_relation =  
    "or" ^ "else" ^ relation ;  
  
xor_relation =  
    "xor" ^ relation ;
```

META DESCRIPTION FOR ADA

```
choice_link =
    choice { choices } ">" ;

factor =
    primary [ power ! $0 ] ;

factors =
    mul_op factor ;

add_op = <
    "+"
    "-"
    "&" $0 > ;

elsif_part =
    "elsif" ^ expression ^ "then"
    + seq_of_stmts ;

else_part =
    "else"
    + seq_of_stmts ;

tag =
    identifier ":" ;

iteration_clause = <
    while_clause
    for_clause $0 > ;

cases =
    "when" choice { choices } ">"
    + seq_of_stmts ;

declare =
    "declare"
    { + decl }
    { + rep_spec ! }
    { + program_component } ;

when_clause =
    "when" ^ expression ;

qualified_exp =
    name "'" agg_or_exp ;
```

META DESCRIPTION FOR ADA

```
accept_action =  
    "do"  
        + seq_of_stmts  
    @ "end" [ ^ identifier ] ;  
  
condition_link =  
    "when" ^ expression ^ "=>" ;  
  
select_alternative = <  
    accept_alternative  
    delay_alternative  
    terminate > ;  
  
or_clause =  
    "or" [ ^ condition_link ]  
        + select_alternative ;  
  
delay_alternative =  
    delay_stmt  
    [ @ seq_of_stmts ] ;  
  
discrete_ranges =  
    "," ^ discrete_range ;  
  
discrim_spec =  
    [ discrim_link ^ ] expression ;  
  
discrim_specs =  
    "," ^ discrim_spec ;  
  
generic_name = <  
    name  
    "<>" > ;  
  
generic_discrete =  
    "(" " "<>" ")" ;  
  
generic_integer =  
    "range" ^ "<>" ;  
  
generic_float =  
    "delta" ^ "<>" ;  
  
generic_fixed =  
    "digits" ^ "<>" ;
```

META DESCRIPTION FOR ADA

```
generic_actual_param = <
    expression
    name
    subtype_indication > ;

index =
    name ^ "range" ^ "<>" ;

indices =
    "," ^ index ;

component_list = <
    components
    null_comp > ;

enum_lit = <
    identifier
    char_lit > ;

enum_lits =
    "," ^ enum_lit ;

relational =
    rel_op ^ simple_exp ;

in_range =
    simple_exp [ ^ "not" ] ^ "in"
    ^ range_or_subtype ;

choice = <
    simple_exp
    discrete_range
    "others" > ;

choices =
    "|" ^ choice ;

power =
    "*" primary ;

mul_op = <
    "*"
    "/"
    "mod" $0
    "rem" $0 > ;
```


META DESCRIPTION FOR ADA

```
primary = <
    decimal_number
    name
    nested_exp
    based_number $0
    enum_lit $0
    char_string $0
    func_call $0
    "null" $0
    aggregate $0
    allocator $0
    type_conversion $0
    qualified_exp $0 > ;
```

```
while_clause =
    "while" ^ expression ;
```

```
for_clause =
    "for" ^ identifier ^ "in" [ ^ "reverse" ! ]
    ^ discrete_range ;
```

```
agg_or_exp = <
    aggregate
    nested_exp > ;
```

```
accept_alternative =
    accept_stmt
    [ @ seq_of_stmts ] ;
```

```
terminate =
    "terminate" ";" ;
```

```
discrim_link =
    name { names } ^ ">" ;
```

```
components =
    { @ component_decl }
    [ @ variant_part ] "" ;
```

```
null_comp =
    "null" ";" ;
```

```
char_lit =
    " " " " " " ;
```

META DESCRIPTION FOR ADA

```
rel_op = <
    "="
    "/="
    "< "
    "<="
    "> "
    ">=" > ;

range_or_subtype = <
    range
    subtype_indication > ;

decimal_number =
    integer [ decimal_part ! $0 ]
        [ exponent ! $0 ] ;

nested_exp =
    "(" expression ")" ;

based_number =
    integer "#" based_integer [ based_decimal ! ]
        "#" [ exponent ! ] ;

allocator =
    "new" ^ name [ ^ allocation ] ;

type_conversion =
    name "(" expression ")" ;

component_decl =
    id_list ":" ^ object_type [ ^ initial ] ";" ;

variant_part =
    "case" ^ name ^ "is"
        { + variant_case }
    @ "end" ^ "case" ";" ;

integer =
    '09' { '09!_' } ;

decimal_part =
    "." integer ;

exponent =
    "E" [ sign ] integer ;
```

META DESCRIPTION FOR ADA

```
based_integer =  
    '09|AZ|az' { '09|AZ|_|az' } ;
```

```
based_decimal =  
    "." based_integer ;
```

```
allocation = <  
    nested_exp  
    aggregate  
    discrim_constraint  
    index_constraint > ;
```

```
variant_case =  
    "when" ^ choice { choices } ^ "=>"  
    + component_list ;
```

```
sign = <  
    "+"  
    "-" > ;
```

META DESCRIPTION FOR ADA0

APPENDIX III META DESCRIPTION FOR ADA0

This appendix contains the META description for the \$0 subset of the previous ADA description. It was the subset implemented in the prototype compiler for the environment.

```
compilation =  
    compilation_unit ;
```

```
compilation_unit = <  
    proc_body > ;
```

```
proc_body =  
    proc_spec ^ "is"  
        { + decl }  
        { + program_component }  
    @ "begin"  
        + seq_of_stmts  
    @ "end" [ ^ identifier ] ";" ;
```

```
proc_spec =  
    "procedure" ^ identifier ;
```

```
decl = <  
    object_decl > ;
```

```
program_component = <  
    proc_body > ;
```

```
seq_of_stmts =  
    stmt  
    { @ stmt } ;
```

```
identifier =  
    'AZ|az' { '09|AZ|_|az' } ;
```

```
object_decl =  
    id_list ":" [ ^ "constant" ] ^ object_type  
        [ ^ initial ] ";" ;
```

```
stmt =  
    simple_stmt ;
```

META DESCRIPTION FOR ADA0

```
id_list =
    identifier { identifiers } ;

object_type = <
    subtype_indication > ;

initial =
    ":" ^ expression ;

simple_stmt = <
    assignment_stmt
    if_stmt
    loop_stmt
    proc_call > ;

identifiers =
    "," identifier ;

subtype_indication =
    name ;

expression = <
    relation
    and_comp
    or_comp > ;

assignment_stmt =
    name ^ ":" ^ expression ";" ;

if_stmt =
    "if" ^ expression ^ "then"
        + seq_of_stmts
    { @ elsif_part }
    [ @ else_part ]
    @ "end" ^ "if" ";" ;

loop_stmt =
    [ iteration_clause ^ ] "loop"
        + seq_of_stmts
    @ "end" ^ "loop" ";" ;

proc_call =
    name ";" ;

name = <
    identifier > ;
```

META DESCRIPTION FOR ADA0

```
relation =
    simple_exp [ ^ relation_part ! ] ;

and_comp =
    relation ^ { and_relation } ;

or_comp =
    relation ^ { or_relation } ;

elsif_part =
    "elsif" ^ expression ^ "then"
        + seq_of_stmts ;

else_part =
    "else"
        + seq_of_stmts ;

iteration_clause = <
    while_clause > ;

simple_exp =
    [ unary_operator ! ] term { terms } ;

relation_part = <
    relational > ;

and_relation =
    "and" ^ relation ;

or_relation =
    "or" ^ relation ;

while_clause =
    "while" ^ expression ;

unary_operator = <
    "+"
    "-"
    "not" > ;

term =
    factor { factors } ;

terms =
    add_op term ;
```

META DESCRIPTION FOR ADA0

```
relational =  
    rel_op ^ simple_exp ;
```

```
factor =  
    primary ;
```

```
factors =  
    mul_op factor ;
```

```
add_op = <  
    "+"  
    "-" > ;
```

```
rel_op = <  
    "="  
    "/="   
    "< "  
    "<="   
    "> "  
    ">=" > ;
```

```
primary = <  
    decimal_number  
    name  
    nested_exp > ;
```

```
mul_op = <  
    "*"   
    "/" > ;
```

```
decimal_number =  
    integer ;
```

```
nested_exp =  
    "(" expression ")" ;
```

```
integer =  
    '09' { '09|_' } ;
```

APPENDIX IV

SYSTEM USER MANUAL

IV.1. INTRODUCTORY MANUAL

This manual contains a brief introduction to how to use the syntax directed editor environment. It does not go into a lot of detail about the individual tools, but tries to give enough information to get a user started into the system. After reviewing the information contained in this manual the user should have enough knowledge to begin using the environment. It is strongly recommended that the user review the user manuals for the individual tools before attempting any major projects. These manuals appear immediately following this introductory manual. This insures an understanding of all of the available features of the tools and permits the optimal utilization of the environment.

PREPARING THE ENVIRONMENT

There are a few things which need to be done before the editor is entered to set up the initial environment in which the editor will work. These involve the two preprocessors, config and meta.

Config is responsible for configuring the editor so that it is comfortable for you to use. It is invoked by typing :

```
config *
```

The "*" is optional. It specifies that any existing configuration (stored in a file called "terminal.tdf") be erased. After the first configuration you probably want to leave the "*" off of the command, or else you will have to reenter all of the information.

When invoked, config will proceed to ask you a number of questions. A "Y" or "y" is a "yes" response, anything else is considered a "no". Your response to a yes or no question is not echoed to your screen, but more

SYSTEM USER MANUAL

questions will appear depending on your response. The first few questions are self explanatory.

A "yes" response to changing the input commands drops you into another series of questions. The name of each command will appear, along with the previous definition if it exists. You must enter something for each command, even if you don't think you will be using it. Each command must be unique, and must begin with a non-printable character. The commands you enter will be echoed to the terminal. Non-printable characters will echo as a "^" followed by the code shifted to a printable character. Thus a control-X will be echoed as "^X". The commands are described in more detail in the SYNDE user manual.

The next section of config asks about automatic output configuration. This is something left over from an earlier version and can simply be answered "no".

The final section deals with the output commands. Most of these (the ones marked with "***") are of no importance unless you are on an unknown type of terminal. Simply hit a return for all of those entries. The three important ones to enter are :

- display tab : the character string used to display a tab. Probably a number of spaces.
- mark elided material : the symbol you want displayed in place of an elided subtree. "+" is a good choice, as is an appropriate string
- divide window : a single character which is used to fill the window header, the suggested one is "-"

Once the terminal is configured to your satisfaction you should check to see that the language syntax description is available for the editor to use. This is a file called "something.sdf" (for ADA it would be "ada.sdf"). If this file exists you are ready to start. If not, type :

SYSTEM USER MANUAL

`meta ada`

or whatever the name of the language is. When it is finished running the ".sdf" file should be present. Any problems encountered here can probably be solved by looking at the META reference manual, or talking to the person responsible for maintaining the environment.

INSIDE THE EDITOR

Once all of the preprocessing is done you are ready to enter the editor. It is invoked by its name, `synde`, followed by the file to be edited and the language to be used. To create an ADA program in a file called "test" you would type :

`synde test ada`

The language name can be omitted if the file already exists from a previous use of the editor. (Don't try to edit files created by different types of editors. They will not work.)

Once you are in the editor you can begin creating the program. The window header tells you the name of the syntax tree node you are at, and it is displayed as a highlighted section on the screen. By moving down the syntax tree (using the down command) you can arrive at the leaves where you enter information or make selections. Moving up, down, left, or right will be movements in the syntax tree and the results will be visible on the screen. Remember that these are all tree movements, not screen movements.

The editor will automatically generate as much of the syntax tree as it can, but it usually can not go far without some help from you. There are three major types of nodes that require your input : 1) conditionals, where you have to tell the editor to insert that node; 2) alternations, where you make a choice about the type of thing to insert; and 3) sets, another form of alternation.

SYSTEM USER MANUAL

Conditional nodes are those which appear surrounded by braces ("[" and "]", or "{" and "}"). These are items which are not needed by the simplest of programs, but may be needed for yours. Conditional nodes are added to the tree by typing the first letter of the name in braces. Once it has been selected the braces will be removed.

Alternations are where you select one of a list of alternatives. As long as help is turned on (the default is on), a list of choices available to you will appear at the bottom of the screen. To make a selection you type enough of your choice to be unambiguous. The editor helps you by doing command completion wherever possible. Each letter you type restricts the available choices, and the current choices are displayed. The editor fills in as much of the command as possible, and when only one possibility remains it accepts that. The concept should become very clear the first time you try it.

The third type of node requiring your actions is the set. This represents a place where a single character is to be entered. It is displayed in a conventional shorthand notation. Thus 'AZ' means any letter between capital A and capital Z, and '09|AZ|az' means any digit or letter. Selection of a set element is done by simply typing the character desired.

That concludes this brief summary of how to use the syntax directed editor environment. Movement to the other tools of the environment from the editor is possible simply by invoking the appropriate commands, but these will not be discussed here. Once again the user is encouraged to read the manuals for each tool to obtain a more detailed understanding of what happens in each one.

SYSTEM USER MANUAL

IV.2. SYNTAX DIRECTED EDITOR MANUAL

This manual covers the functions and commands of the syntax directed editor. It includes information on how to enter the editor, a summary of all of the commands, and how to create the program.

ENTERING THE EDITOR

The editor is invoked by its name, `synde`, followed by the name of the file to be edited. If the file does not exist, the filename must be followed by the name of the language to be used. To create a new ADA program in a file called "test" you would type :

```
synde test ada
```

To edit an already existing file called "test" the command is simply :

```
synde test
```

Inclusion of the language name at the end of the command will cause no adverse effects if the file already exists; it will simply be ignored by the editor. All of the files to be edited must be created through the editor. Editing files created by other forms of editors or creation commands will not work properly.

EDITOR COMMANDS

The particular set of keystrokes used to invoke a particular command is determined by the user during the system configuration. (See the Config user manual for details.) This section discusses what each command does when invoked. Each command will respond with either a change in the current display or an appropriate message to the display.

Movement within the editor is based on movement within the syntax tree. The "focus" is the part of the syntax tree which is currently being dealt with by the editor. The focus is the root of the subtree being considered by the editor. The focus image is displayed in the main window of the display screen by some form of highlighting, usually reverse video. The name of the node will appear

SYSTEM USER MANUAL

in the window header. Most of the movement commands are movements to adjacent nodes such as parent, child, or sibling.

The MOVE RIGHT command moves the focus to the right sibling of the focus. If no right sibling exists, the tree is ascended until some ancestor is found which has a right sibling. This right sibling then becomes the focus. This corresponds to an inorder traversal of the tree.

The MOVE LEFT command moves the focus to the left sibling of the focus. If no left sibling exists, the tree is ascended until some ancestor is found which has a left sibling. This left sibling then becomes the focus. This corresponds to a reverse of an inorder traversal of the tree.

The MOVE LONG RIGHT command moves the focus to the right and past any siblings which are generated from the same repetition element in a production.

The MOVE LONG LEFT command moves the focus to the left and past any siblings which are generated from the same repetition element in a production.

The MOVE UP command moves the focus to the parent of the focus, if it exists. This is equivalent to examining a higher syntactical level. If the parent has children other than the current focus, this will expand the current area of interest, forcing an expansion of the highlighted area on the screen. This may be viewed as a "zoom-out" command.

The MOVE DOWN command moves the focus to the leftmost child of the focus if one exists. This corresponds to a lower, or more specific, syntactical level. A reduced viewing area, or "zoom-in" will occur if the focus has more

SYSTEM USER MANUAL

than one child.

The MOVE LONG UP command is equivalent to a series of move up commands until the highlighted display area is expanded. It corresponds to moving up the syntax tree until a node with more than one child is encountered.

The MOVE LONG DOWN command is equivalent to a series of move down commands until the highlighted area is reduced. This corresponds to moving down the syntax tree until a node with more than one child is encountered, and selecting the left child of that node.

The MOVE TO LEAF command descends the tree from the current focus in an inorder traversal until a leaf node is encountered. This provides a quick way of moving to the bottom of the tree.

The editor stores the location of the most recent focus prior to the current focus. The MOVE TO LAST FOCUS command moves the focus to the stored focus.

The program tree also contains storage space for up to ten markers, or pointers into the syntax tree. Markers zero through four may be set by the user, while markers five through nine are reserved by the system for marking errors from other tools in the environment. The MARK command may be used to clear an existing marker at the focus, or to set any of the user markers, zero through four, to the current focus. The GO command may be used to move the focus to any of the set markers, the root of the program tree, or the root of the clipping tree. Markers are preserved between editing sessions.

Those are the commands for moving the focus within the current tree.

SYSTEM USER MANUAL

The next set of editing commands deal with operations which change the syntax tree.

The INSERT RIGHT command inserts a conditional element as the right sibling of the focus. A valid conditional element must be capable of being inserted at that location. This is determined by the production definition of the parent node, which also determines the type of element to be inserted. An optional element may be inserted only if it does not already exist. Insertion of consecutive identical, unestablished repeaters is not allowed by the editor, because it is considered a useless, though valid, operation. Insertion is done only for a element which is the immediate right sibling of the focus. Thus it may be necessary to establish certain unneeded conditionals when two or more successive conditionals occur in order to move the focus to the proper place. These unneeded elements may be removed once the desired element has been inserted.

The INSERT LEFT command is identical to the insert right command except that it establishes a conditional element as the left sibling of the focus.

The CLIP command copies the subtree designated by the focus to a "clipping tree". The previous clipping is discarded, and the program tree remains unchanged. This corresponds to the "cut" of a "cut and paste" operation. The clipping is retained until another clipping occurs. Changes may be made to the clipping tree by using the go command to move to the root of the clipping tree and then editing that tree. Clipping may occur at any node except a leaf. This is to prevent inadvertent loss of a clipping tree. A leaf node is a degenerate subtree and is thus easily regenerated.

The COPY command attaches a copy of the clipping tree to the program tree at the focus. The previous subtree of the focus is irrecoverably lost. This

SYSTEM USER MANUAL

corresponds to the "paste" portion of the "cut and paste". The syntactic type of the clipping tree root must match that of the focus, or, if the focus is an alternation, of one of the alternatives. In the second case the clipping tree is copied as the child of the focus. The syntax type appears as the name in the window header. The clipping tree type may be obtained by displaying the clipping in the second window.

The KILL command deletes the focus and its subtrees from the program tree, if possible. If the focus is a conditional element the node and any subtrees are simply deleted. If the node is not a conditional it must be retained for syntactic correctness. In that case all of the children and subtrees of the focus are deleted, and if it is a concatenation, any automatic expansions are performed. This generates a "clean" node, exactly as if this was the first occurrence of that node. The subtrees lost in a kill operation cannot be recovered. If possible recovery is desired the delete command should be used.

The DELETE command corresponds to a clip operation immediately followed by a kill. The effect is the same as that of a kill except that the deleted tree is stored in the clipping tree. The old tree may be recovered by doing a copy command at the focus. The delete operation is equivalent to a kill if the focus is a leaf. Deletions while editing the clipping tree are not allowed.

The next three operations are not commands of themselves, but are the means of altering the leaf nodes. The first of these is **CONDITIONAL NODE ESTABLISHMENT**. Conditional nodes are those which appear on the display surrounded by braces ("[" and "]", or "{" and "}"). Alternatives and sets are established simply by making a selection as discussed below. For a concatenation, typing the first character of the displayed name establishes the

SYSTEM USER MANUAL

node. Once a node is established the braces are removed. If the node is a repeater, another unestablished repeater of the same type is inserted immediately to the right of the focus.

ALTERNATIVE SELECTION occurs when a leaf node has an alternation definition and the user must select one of the available alternatives. Selection is accomplished by typing the name of the desired selection, with command completion by the editor. Command completion involves the filling in of as much of the command as possible by the editor. This reduces the number of keystrokes required and speeds the selection process.

Sets are a form of alternation where the possible alternatives are single characters. SET SELECTION is accomplished by the typing of the desired character. A conventional shorthand is used for sets. Thus 'AZ' means any character between capital A and capital Z, while '09|AZ|az' represents any letter or digit. When a set element is selected it replaces the set name on the display.

The final set of commands are those which provide the user with control over the display of the program and provide the interface to other tools in the programming environment.

The HELP command toggles the user request for help menus. The default is for help to be turned on. The only menu currently available shows a list of alternatives available when the focus is at an alternation node.

The ELIDE command toggles the elide flag for the focus node. When turned on, the elide flag limits the extent of the tree display. While in the elided section of the tree, no portion of the tree outside of the elided section will be displayed. When the focus moves up the tree past the elided node, the

SYSTEM USER MANUAL

display will suppress display of that node and it's subtrees and replace it with the string provided by the user during configuration. This feature is useful in indicating levels of modularity within the program.

The WINDOW command controls the presence of a second window on the screen. When present the window can be used to display the clipping tree, or any of the subtrees indicated by markers zero through nine. The second window is for display purposes only; no editing will be done in this window. However any changes made in the main window will appear in the second window if they affect the subtree displayed there. The second window may be opened or closed at any time. The header for the second window is the name of the root of the subtree being displayed.

The REDRAW SCREEN command clears out the current screen and reproduces the current correct view. This is useful if transmission errors cause the screen display to become garbled.

The SAVE FILE command causes the program and any recent changes to be written to the file from which the program was originally obtained. This allows the user to save changes prior to the automatic save which is performed when the editor is exited.

The WRITE FILE command allows the writing of the program currently being edited to a file of another name. If the specified file already exists, the user is given the option of deleting the old file.

The INVOKE COMPILER command causes the language specific compiler for the program being edited to be loaded and executed. The compiler flags any program errors using markers five through nine, and halts when compilation is complete or all the markers have been used. After compilation, execution

AD-A138 009

A SYNTAX DIRECTED EDITOR ENVIRONMENT(U) AIR FORCE INST
OF TECH WRIGHT-PATTERSON AFB OH SCHOOL OF ENGINEERING
J R KOSLOW 05 DEC 83 AFIT/GCS/MA/83D-3

2/2

UNCLASSIFIED

F/G 9/2

NL

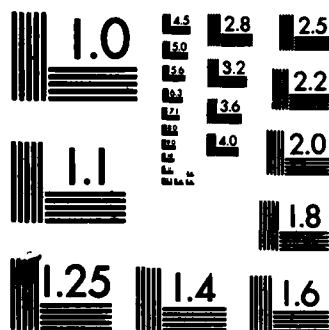
END

FILED

DEC 83

AFIT

GCS



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

SYSTEM USER MANUAL

returns to the editor and the focus is set to error marker five, if set, or to the program tree root.

The INVOKE INTERPRETER command causes the loading and execution of the language interpreter. If errors are encountered, return to the editor is exactly like that of the compiler. See the specific manuals for interpreter commands.

The INVOKE LISTER command causes loading and execution of the language independent program lister. This generates a text image of the program tree being edited which is suitable for human reading or as input to conventional, text-based environment tools such as compilers. Unestablished conditional nodes are not printed as these represent unnecessary, unincluded nodes. Unexpanded required nodes are shown by the node name enclosed in angle brackets ("<" and ">"). Elided subtrees are not suppressed in the generated image. Control returns to the editor when listing is complete.

The INVOKE SYNDE command causes the syntax directed editor to be invoked for a new file. This is often referred to as visiting another file. The user is prompted for the file name and language.

The EXIT EDITOR command terminates the editing session. The program tree is saved, system cleanup occurs, and the session is terminated.

IV.3. TERMINAL CONFIGURATION MANUAL

The terminal configuration program, called config, is used to adapt the editor to a particular user. It is also used to obtain information about the terminal from the user, particularly if the terminal is an unknown or uncommon model. The program interactively prompts the user for the required information, and stores the obtained information in a terminal description file.

SYSTEM USER MANUAL

Config is invoked by typing :

```
config *
```

where the "*" is optional. Inclusion of the "*" indicates that the current terminal description file (terminal.tdf) is to be cleared before running the config program. The user is then given the opportunity to enter new information or modify the current information.

Config will ask a number of questions which will require a yes or no answer. An input of "Y" or "y" is taken as a "yes", any other key being interpreted as a "no".

The first information requested by config deals with the terminal's lines per screen, characters per line, and number of lines to be used in the second window. The prompt will include that value and an inclusive range of valid values. Simply type the new value, or a carriage return to retain the old value. Invalid values are rejected and reprompted.

The next set of prompts deal with the input command sequences. Details of each command's functions are available in the Synde manual. Config presents the old sequence (if it exists) and the option to change it or proceed to the next item. All input commands must exist and must be unique. They should begin with a non-printable character. To enter a command sequence simply type the sequence of keys desired to invoke that function. As each key is hit it is echoed to the display. Control characters are echoed as ^x where x is the control character plus an offset of 64 to obtain a printable character. Thus a line-feed (ASCII value 10) will be displayed as ^J. Config presents a number of opportunities to reexamine the sequences so mistakes can be corrected with another pass through the input commands.

The next question asks about automatic output configuration. This is a

SYSTEM USER MANUAL

carryover from a previous version and is only of interest if you would like to see what the escape sequences are for some of the output commands. Usually the answer to this question should be "no".

Finally config will request information about the display output sequences. Most of these do not need to be entered unless you are on a terminal whose type is not known to the system. The editor only refers to those sequences if it does not know what to do. Those entries are marked below, and in the config program, with "**" at the beginning of their name. All of the output sequences are entered and echoed exactly like the input command sequences.

The sequences prompted for are :

- ****initialize terminal** : A sequence sent to the terminal at the start of an editing session to allow for any special startup.
- **display tab** : Used for indentation in the program. Usually a number of spaces.
- **mark elided material** : Used to represent an elided program subtree in the display.
- **divide windows** : a single character used to fill the header line of each window
- ****clear screen** : Clear the terminal screen
- ****position cursor** : Prefix of command to position the cursor on the screen. This sequence is followed by the display line plus 32 and display column plus 32. Other forms of cursor movement are handled for the known terminals but cannot be manually entered.
- ****erase to end of line** : Clear the display from the cursor location to the end of the line.
- ****enter reverse video mode** : Used to set the terminal into a highlighting mode to distinguish the focus.
- ****exit reverse video mode** : exit the highlighting mode set by "enter reverse video mode"
- ****terminate terminal** : Sent to the terminal at the end of an editing session to handle any special terminal cleanup.

IV.4. META PREPROCESSOR

This manual covers material dealing with the Meta preprocessor which transforms a textual syntax description into a syntax description which can be used by the tools in the environment. The short discussion of how to use the preprocessor is followed by a rather lengthy discussion of how to create the textual syntax description. Those users who are interested only in using available language descriptions need only read the first section.

RUNNING THE META PREPROCESSOR

The Meta preprocessor is invoked by a command of the form :

```
meta filename subset_index
```

where filename.syn is the name of a textual meta syntax description. The subset_index is an optional argument indicating the subset elements to be eliminated from the description. Meta creates a syntax description file of the name "filename.sdf" where filename is the name given in the argument list, possibly extended by digits from the subset_index. For example the command:

```
meta ada
```

will create the syntax description ada.sdf from the textual description ada.syn.

The subset_index argument is a "#" followed by a series of digits in the range zero through seven. This indicates that any element in the textual description marked with the indicated subset numbers be eliminated from the syntax description generated by Meta. (See page 95 for more information on language subsets.) The command :

```
meta ada #0
```

would create the syntax description file ada0.sdf from the textual description ada.syn while removing those elements marked with subset zero.

SYSTEM USER MANUAL

LANGUAGE DESCRIPTION

The textual syntax description input for Meta is presented as a sequence of production rules, each of which defines a non-terminal of the language. The first production rule must define the non-terminal which represents the language goal symbol. Appendix I specifies the format to be followed in a Meta description.

Each production rule is of one of two forms : a concatenation or an alternation. A concatenation is an ordered sequence of elements which specify a template to be laid into the program tree structure beneath a node of that type. Individual elements in a concatenation list may be conditionals. Conditionals are either options (appearing zero or one time) or repeaters (appearing zero or more times). Options are enclosed in square brackets ("[" and "]"), while repeaters are enclosed in braces ("{" and "}"). Both may contain only one element inside the delimiters. The hide indicator ("!") may be used to mark a conditional element so that it is not automatically synthesized into the syntax tree. This is useful for elements of the language which are rarely used. Each concatenation must contain at least one unconditional element to be synthesized into the syntax tree.

An alternation is a list of alternatives, only one of which may be selected. Each alternative must be a single unconditional element. The alternative list is surrounded by angle brackets ("<" and ">") to distinguish it from a concatenation.

Each syntactic element of the definition may be a non-terminal identifier, a literal string enclosed in quotes, or a set construct. Each non-terminal must be defined exactly once in the syntax description. Literal strings are used to represent the reserved words and delimiters used by the language. The set

SYSTEM USER MANUAL

construct is typically used in specifying identifiers or numbers whose individual character components are not specified until the synthesis process.

A set construct represents a compact way of showing an alternation whose alternatives are single characters. Elements in the set may be single characters or pairs of characters representing an inclusive range in the ASCII character set. Thus 'AZ' represents any letter between capital A and capital Z, while '09|AZ|az' represents any letter or digit. Meta requires that the set alternatives be presented in ascending ASCII order.

FORMAT CONTROLS

In addition to the syntax description of the language, the editor and other tools require information about the format to be used when displaying the program or generating a text file of the program. Thus the Meta description includes format controls of three types : space marks, newline marks, and indentation.

A space mark ("^") preceding or following an element results in the placement of a corresponding space in the program tree image. A newline mark ("@") preceding an element causes a new line to be generated on the display followed by the proper number of tabs for the current level of indentation. An indentation mark ("+") preceding an element indicates that the current indentation level is to be increased and causes a new indented line to be generated. All elements in the subtree below the node corresponding to that element will be indented, after which the former indentation level is restored.

Format controls for an element take effect only when the element is synthesized into the program tree. Format controls for conditional elements must be placed carefully to insure a desirable appearance whether the conditional element is present or not.

SYSTEM USER MANUAL

LANGUAGE SUBSETS

Alternative and conditional elements in the Meta description may be marked with a subset indicator. This indicator is a dollar sign ("\$\$") followed by a series of digits in the range zero through seven ("0" to "7"). Each digit indicates a subset in which the marked element will not appear. This can be used to restrict the use of certain constructs, or reduce the complexity of the language available. These applications are particularly useful in an academic or other learning environment.

The Meta preprocessor can be instructed to omit the subset elements by using the `subset_index` argument in the invocation command. When all references to a non-terminal are removed, its production rule is no longer required. All such unreferenced rules are eliminated from the syntax description to conserve space. Appendix III contains the \$0 subset of the description in Appendix II.

DESIGNING A META LANGUAGE DESCRIPTION

Creating a Meta description of a language can be a difficult and time consuming process. Reference to an existing definition of the language, particularly an extended BNF description, can substantially reduce the effort required. This section discusses the major differences between extended BNF and Meta, and the changes needed to design the Meta description.

Meta requires a simpler form of expression than is generally available in extended BNF. In particular, Meta disallows the use of complex expressions within options, repetitions, or alternations. In many cases a new non-terminal must be introduced in Meta to replace these complex expressions. For example, the ADA reference manual contains the following extended BNF definition for a term :

```
term ::=
```

SYSTEM USER MANUAL

```
factor { multiplying_operator factor }
```

Having two elements inside the repeater (multiplying_operator and factor) is not allowed in a Meta description. A corresponding Meta description would be :

```
term =  
    factor { factors };  
  
factors =  
    mul_op factor ;
```

Most of the problems encountered in transforming extended BNF to Meta are of this nature.

The design of the syntax description for a syntax directed editor environment requires the designer to address issues usually not of concern in the syntax description. Because the editor enforces a standard format for program display, format control characters must be added to the syntax description in such a way as to provide a desirable display whether all conditionals are present or not. The description must also be written to provide a reasonable human interface. One area of concern involves conditional elements and alternatives, elements which require user interaction. These represent decision points for the user and as such should be minimized, or at least placed in a position where a decision seems natural.

If an alternation contains an element which is itself an alternation the user is required to make two successive decisions. This can be reduced to a single decision by including each alternative of the second element as an alternative of the first. Applying this to the ADA reference manual definition for primary and literal :

```
primary ::=  
    literal | aggregate | name | allocator  
    | function_call | type_conversion  
    | qualified_expression | (expression)
```

SYSTEM USER MANUAL

```
literal ::=
    numeric_literal | enumeration_literal
    | character_string | null
```

it reduces to the equivalent Meta definition :

```
primary = <
    decimal_number name nested_expression
    based_number enum_lit char_string
    func_call "null" aggregate allocator
    type_conversion qualified_expression > ;
```

by including the alternatives for literal, as well as those of numeric_literal, within primary. Also note that the alternatives have been rearranged in an attempt to name the more frequently used ones first, as this is the order they appear to the user.

Another design concern is the selection of non-terminal names. These names will often appear in incomplete code fragments indicating the type of object to be placed in that location. The need for names with mnemonic value must be balanced against a reasonable length to avoid cluttering the screen.

The limitations required by Meta for the descriptions it handles can cause complications in the editing process. For example the ADA reference manual defines an identifier as :

```
identifier ::=
    letter { [underscore] letter_or_digit }
```

which eliminates the possibility of double or trailing underscores. An equivalent Meta definition obtained by removing the complex expression is :

```
identifier =
    'AZ|az' { score_digit_letter } ;

score_digit_letter =
    ["_"] '09|AZ|az' ;
```

This requires an additional keystroke for each subsequent letter or digit. An alternative Meta definition to eliminate the extra production and provide a

SYSTEM USER MANUAL

smoother format for identifier entry is :

```
identifier =  
    'AZ|az' { '09|AZ|_|az' } ;
```

However this allows illegal double or trailing underscores which must be detected by the compiler. The added ease of identifier entry, a common occurrence, was considered significant enough to warrant deferring detection of such trivial errors to the compiler phase (Reference Ferguson). Fortunately most of the design decisions that need to be made do not require compromising the syntactic validity of the programs produced.

GLOSSARY

APPENDIX V GLOSSARY

- BNF** Backus-Naur Form, a commonly used method of describing a language syntax
- Conditional node** a syntax tree node whose presence is not required for syntactic correctness. Includes options and repeaters.
- Editing focus** the subtree being examined by the user at a particular time
- Establish a node inclusion of a conditional node into the tree structure**
- Extended cursor** means of indicating an editing focus even if it is larger than one character. Usually accomplished by highlighting the editing focus
- Format controls** symbols of a META description which control the display of the syntax tree in a form which is recognizable to the user
- Frontier** the leaves of the syntax tree
- Hidden element** a conditional element of the syntax description which is not automatically shown to the user
- Highlight mode** a means of setting a certain section of a display so that it is different than the surrounding display. Often accomplished by switching the background color for that section
- Input commands** the keystroke sequences used by the user to invoke the commands of the editor
- Language goal symbol**
the non-terminal symbol which is the starting symbol of the language syntax definition
- Modular elision** specifying levels of modularity within the syntax tree. The entire subtree below the elided node is treated as a single unit, and is displayed as such
- Option element** syntax description element which may appear zero or one time
- Output command sequence**
set of keystrokes sent to the terminal to accomplish the various type of screen changes
- Production** a rule of a syntax description describing the expansion of a non-terminal element

GLOSSARY

Program tree synthesis

the creation and modification of a syntax tree as determined by the syntax definition and user actions

Repeater element

syntax description element which may appear zero or more times

Software development environment

the collection of interconnected tools within which the user develops computer software

Syntax description

a textual description (often BNF or Meta) of the rules which govern the syntax of a computer language

Syntax directed editor

a tool for entering a program into the computer which automatically generates correct program syntax based on the language being used

Syntax tree

tree representation of a program. Nodes of the tree represent the terminal and non-terminal symbols of the syntax description

User friendly

easy to understand and use by anyone with reasonable cause to be using the tool

VITA

John R. Koslow was born 5 June 1960 in Biloxi, Mississippi. He graduated as class valedictorian from Hayfield High School in Alexandria, Virginia in 1978. He graduated from Carnegie -Mellon University in 1982 with a degree of Bachelor of Science in Mathematics (Computer Science) / Economics. He received a commission in the USAF through the ROTC program. His first assignment, in June of 1982, was to the School of Engineering, Air Force Institute of Technology where he began work as a graduate student in computer science.

Permanent address: 546 Hallock Street
Pittsburgh, PA 15213

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED			1b. RESTRICTIVE MARKINGS	
2a. SECURITY CLASSIFICATION AUTHORITY			3. DISTRIBUTION/AVAILABILITY OF REPORT APPROVED FOR PUBLIC RELEASE DISTRIBUTION UNLIMITED	
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE				
4. PERFORMING ORGANIZATION REPORT NUMBER(S) AFIT/ENG/DA/83D-3			5. MONITORING ORGANIZATION REPORT NUMBER(S)	
6a. NAME OF PERFORMING ORGANIZATION School of Engineering		6b. OFFICE SYMBOL (If applicable) AFIT/ENG	7a. NAME OF MONITORING ORGANIZATION	
6c. ADDRESS (City, State and ZIP Code) Air Force Institute of Technology Wright-Patterson AFB, OH 45433			7b. ADDRESS (City, State and ZIP Code)	
8a. NAME OF FUNDING/SPONSORING ORGANIZATION		8b. OFFICE SYMBOL (If applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER	
8c. ADDRESS (City, State and ZIP Code)			10. SOURCE OF FUNDING NOS.	
11. TITLE (Include Security Classification) See Box 19			PROGRAM ELEMENT NO.	PROJECT NO.
			TASK NO.	WORK UNIT NO.
12. PERSONAL AUTHOR(S) Koslow, John Richard				
13a. TYPE OF REPORT MS Thesis		13b. TIME COVERED FROM 83/04 TO 83/12		14. DATE OF REPORT (Yr., Mo., Day) 83 Dec 05
15. PAGE COUNT 107		16. SUPPLEMENTARY NOTATION		
17. COSATI CODES		18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)		
FIELD 09	GROUP 02	SUB. GR.	Syntax directed editor, language oriented editor ADA, Software development environment	
19. ABSTRACT (Continue on reverse if necessary and identify by block number)				
TITLE: A SYNTAX DIRECTED EDITOR ENVIRONMENT THESIS CHAIRMAN: Roie Black, Major, USAF This document describes the implementation and modification of a software development environment for a medium size computer which is based on a syntax directed editor. Although it was developed for use with the ADA programming language, most of the environment is driven by a language syntax description, and can therefore process virtually any programming language. This environment is an extension of a prototype developed previously at the Air Force Institute of Technology.				
20. DISTRIBUTION/AVAILABILITY OF ABSTRACT UNCLASSIFIED/UNLIMITED <input checked="" type="checkbox"/> SAME AS RPT. <input type="checkbox"/> DTIC USERS <input type="checkbox"/>			21. ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED	
22a. NAME OF RESPONSIBLE INDIVIDUAL John A. Koslow, 21st, USAF			22b. TELEPHONE NUMBER (Include Area Code) (513) 255-5533	22c. OFFICE SYMBOL AFIT/EN

FILMED

384